

1 Arm systems semantics

2

3

Ben Simner
University of Cambridge

4

5

6

This thesis is submitted for the degree of Doctor of Philosophy
Department of Computer Science and Technology
Wolfson College, Cambridge



7

8

March 28, 2025

9 **Preface**

10 This dissertation is the result of my own work and includes nothing which is the outcome of work done in
11 collaboration except where specifically indicated in the text.

12 It is not substantially the same as any work that has already been submitted, or, is being concurrently
13 submitted, for any degree, diploma or other qualification at the University of Cambridge or any other
14 University or similar institution except as declared in the preface and specified in the text.

15 It does not exceed the prescribed word limit for the relevant Degree Committee.

16 **Abstract**

17 Computing relies on architecture specifications to decouple hardware and software development. Historically
18 these have been prose documents, with all the problems that entails, but research over the last ten years has
19 developed rigorous and executable-as-test-oracle specifications of mainstream architecture instruction sets
20 and ‘user-mode’ concurrency, clarifying architectures and bringing them into the scope of programming-
21 language semantics and verification.

22 However, the *system semantics*, of address translation and TLB maintenance, instruction-fetch and its
23 required cache maintenance, and exceptions and interrupts, remains mostly obscure, leaving us without a
24 solid foundation for verification of security-critical systems software.

25 We develop precise mathematical models, for those aspects of the Arm A-class architecture. We implement
26 these models as executable models, in both microarchitectural-flavoured operational and declarative
27 axiomatic style formats. We validate these models, against currently available hardware through the
28 production and evaluation of hardware test harnesses and test suites, and against the architectural intent
29 through discussions with Arm architects. We give a variety of hand-written and machine-generated litmus
30 tests, exercising parts of the architecture previously unexplored.

31 We discuss the nature of developing such models, and the challenges that writing specifications of existing
32 systems entails. We briefly touch on how these models have evolved over time, and how we imagine they
33 will evolve in the future as the remaining questions are resolved.

A lay summary

34

35 Modern computing devices have become increasingly complex over time, powered by chips with many
36 interacting components: network controllers; audio and signal processing; graphics processing; memory
37 controllers; and at the heart is the central processing unit, or CPU. The CPU is the principal director of
38 the entire machine, in charge of running all the software and coordinating all the components together.
39 The CPU does calculations, directs data to be read or written, manages connections over the network,
40 and receives all the input from the outside world to decide how to respond.

41 Historically, ‘computers’ were mechanical calculators. As time progressed, computers progressed from
42 mechanical to electronic, and became general computation devices and not just simple calculators. They
43 moved from manual to automated, not actioning operations as directed by a human but executing *programs*:
44 pre-determined sequences of *instructions* telling the computer which set of operations to perform.

45 Machines like *EDSAC* paved the way for the modern computer. Its instructions were few and simple, e.g.:
46 read a number from the input tape, add or subtract two numbers, load a number from memory, store a
47 number into memory, go to a particular instruction, display a number to the user, stop the machine and
48 ring the bell. Computers have advanced much since then, but the interface has not: we still program by
49 giving the machine a sequence of very simple instructions to perform. Today there are many *instruction set*
50 *architectures* (ISAs), each defining standard collections of instructions. CPUs with Arm, x86 (Intel/AMD),
51 Power (IBM), and RISC-V ISAs now power billions of devices globally. These ISAs are *far* bigger than
52 *EDSAC*’s meagre ~20 instructions, with the, comparatively small, Arm ISA defining 402 instructions in
53 its base architecture alone.

54 Modern CPUs have advanced in ways other than just having larger ISAs: caches are placed between the
55 CPU and memory, making data much quicker to access; they have become *multicore*, placing multiple
56 CPUs side-by-side on the same chip; and CPU designs have become *superscalar*, able to perform multiple
57 operations at the same time by executing instructions in a pipelined fashion.

58 Superscalar and multicore machines can therefore have multiple instructions touching the same shared
59 data simultaneously. If the programmer instructs two CPUs to concurrently access the same location,
60 then the behaviour of the machine is now determined not by the simple description of the individual
61 instructions, but by the complex interaction of these implementation-specific optimisations. Figuring out
62 *what* can happen in that case is the field of *relaxed memory*, essentially: what happens if the user pushes
63 two buttons that touch the same data at the same time? That is the object of study of this thesis.

64 We are interested in those buttons not for the every day programs, but those which give lower-level control
65 of the machine. These are the instructions used by operating systems to manage many programs at once,
66 and by hypervisors in the cloud to protect and isolate multiple customers virtual machines from one
67 another. Understanding the behaviour of those instructions is crucial in order to be able to develop robust
68 software that uses them and to make credible claims about that software.

69 In particular, we will investigate three areas of a modern high-performance multiprocessor architecture:
70 the ability to change the program being executed on the fly (*self-modifying code*); the machinery used
71 by systems software to control the access programs have to shared data (*virtual memory*); and the way
72 external events interact with a running program, and, in particular, the switch between a program and its
73 supervisor (*exceptions/interrupts*).

74 We do this in three steps. We (1) engage in broad technical discussions with architects and hardware
75 designers. We guide such discussions with specific software patterns or hardware optimisations in mind,
76 focusing on small representative experimental programs or *litmus tests*. We (2) reinforce the results of those
77 discussions through empirical analysis of existing hardware by building tools and using them to gather
78 experimental data, which further informs our and the architects’ understanding. Finally, we (3) employ
79 the tools and techniques from programming language theory to build robust formal mathematical models
80 giving a clear definition to that interface. All this *clarifies* the architects’ intent, gives software engineers
81 a sturdier basis to appeal to, and hopefully will enable richer efforts in certification and verification of
82 those key pieces of software that we all rely on to perform correctly and to keep our data secure.

Contents

83

84	Preface	2
85	Abstract	3
86	A lay summary	4
87	1 Introduction	10
88	1.1 Arm-A architecture overview	11
89	1.2 Systems software	12
90	1.3 Relaxed memory	13
91	1.4 Contributions	14
92	1.5 Publications and collaborations	15
93	1.6 Thesis overview	16
94	2 Modelling Arm: background	17
95	2.1 Relaxed behaviours and litmus testing	17
96	2.1.1 Thread-local ordering	19
97	2.1.2 Coherence	21
98	2.1.3 Multi-copy atomicity	21
99	2.2 Intra-instruction semantics	23
100	2.3 Arm-A operational model	25
101	2.4 Arm-A axiomatic model	27
102	2.4.1 Arm-A candidate executions	27
103	2.4.2 Arm-A axioms	32
104	2.5 The isla-axiomatic tool	34
105	2.5.1 ISA/concurrency interface	36
106	2.5.2 Extended Cat with Sail interface	37
107	I Instruction fetch	39
108	3 Relaxed instruction fetching	40
109	3.1 Introduction	41
110	3.2 Industry practice and the existing Arm prose	42
111	3.3 Modifiable instructions	44
112	3.4 Coherence	46
113	3.4.1 Instruction-to-Instruction coherence	46
114	3.4.2 Data-to-Instruction coherence	47
115	3.4.3 Instruction-to-Data coherence	48
116	3.5 Cross-thread synchronisation	49
117	3.6 Cache maintenance	50
118	3.6.1 Synchronisation on a single thread	50
119	3.6.2 Broadcast cache maintenance	51
120	3.7 Dependencies	53
121	3.7.1 Address dependencies	53
122	3.7.2 Control dependencies	53
123	3.8 Multi-Copy Atomicity	54
124	3.9 More on instruction caches	54
125	3.10 Points of Unification and Coherency	55

126	3.10.1 Late unification	56
127	3.10.2 Promotion	57
128	3.11 Cleans and invalidates are like reads and writes	57
129	3.11.1 Cleans are similar to reads	57
130	3.11.2 IC invalidates are not quite like writes	57
131	3.11.3 DC and IC address speculation	58
132	3.11.4 DC might be to same address	59
133	3.11.5 DC ordering with respect to other memory accesses	60
134	3.12 Same-cache-line ordering	61
135	3.13 Mixed-size instruction fetching	62
136	3.14 Cache type strengthening: IDC and DIC	63
137	3.14.1 IDC	63
138	3.14.2 DIC	63
139	3.15 Related Work	64
140	4 Operational instruction fetching	65
141	4.1 An Operational Semantics for Instruction Fetch	65
142	4.2 iFlat Transitions: a brief summary	66
143	4.2.1 An example: DC/IC cache synchronisation	66
144	4.3 The iFlat state	68
145	4.3.1 Fetch queues	68
146	4.3.2 Abstract instruction caches	68
147	4.3.3 Global abstract data cache	69
148	4.3.4 Outcome types	69
149	4.3.5 Pseudocode states	70
150	4.4 iFlat Transitions: in full	70
151	4.4.1 New transitions	70
152	4.4.2 Updated transitions	72
153	4.4.3 Auxiliary definition – cache line of minimum size	73
154	4.4.4 Handling cache type strengthenings	74
155	5 An axiomatic instruction fetch model	75
156	5.1 Candidates for self-modifying programs	75
157	5.1.1 Explicit events and program order	75
158	5.1.2 Same-location	76
159	5.1.3 Generalised Coherence	76
160	5.1.4 Dependencies	76
161	5.1.5 Reads-from	77
162	5.2 Axioms and auxiliary relations	77
163	5.2.1 Arm ifetch events and relations	77
164	5.2.2 Cache maintenance	77
165	5.2.3 Coherence	80
166	5.2.4 Program order	80
167	5.2.5 Instruction synchronisation (ISB)	81
168	5.2.6 Data synchronisation (DSB)	81
169	5.2.7 Data cache maintenance (DC) is ordered like a read	81
170	5.2.8 Cache maintenance operations and cache lines	81
171	5.2.9 Constrained Unpredictable	82
172	6 Validating the ifetch models	83
173	6.1 The models correctly captures the architectural intent	83
174	6.2 Correspondence between the models	83
175	6.2.1 Making the operational model executable as a test oracle	83
176	6.2.2 Making the axiomatic model executable as a test oracle	84
177	6.3 Equivalence of the models	86
178	6.4 Validating against hardware	86
179	6.4.1 Results from hardware	86

180	II Virtual memory	88
181	7 Pagetables and the VMSA	89
182	7.1 Introduction	89
183	7.2 Virtual Memory	89
184	7.3 Arm Translation Tables	90
185	7.3.1 Translation table format	92
186	7.3.2 Attributes	93
187	7.4 The Arm translation table walk	95
188	7.5 Virtualisation	97
189	7.6 Translation regimes	102
190	7.7 Caching in TLBs	103
191	8 Relaxed virtual memory	105
192	8.1 Virtual memory litmus tests	107
193	8.2 Aliased data memory	109
194	8.2.1 Virtual coherence	109
195	8.2.2 Aliasing different locations	112
196	8.2.3 Might be same (physical) address	112
197	8.2.4 Must not be same physical address	112
198	8.3 What can be cached in TLBs	112
199	8.3.1 Microarchitectural TLBs	113
200	8.3.2 Model MMU	114
201	8.3.3 Invalid entries	115
202	8.4 Reads not from TLB	115
203	8.4.1 Out-of-order execution	115
204	8.4.2 Enforcing thread-local ordering	118
205	8.4.3 Enhanced Translation Synchronization	123
206	8.4.4 Forwarding to the translation table walker	124
207	8.4.5 Speculative execution	125
208	8.4.6 Single-copy atomicity	127
209	8.4.7 Multi-copy atomicity	127
210	8.4.8 Translation-table-walk intra-walk ordering	128
211	8.4.9 Multiple translations within a single instruction	128
212	8.5 Caching of translations in TLBs	132
213	8.5.1 Cached translations	132
214	8.5.2 TLB fills	133
215	8.5.3 microTLBs	133
216	8.5.4 Partial caching of walks	135
217	8.6 TLB maintenance	136
218	8.6.1 Recovering coherence	136
219	8.6.2 Thread-local ordering and TLBI	139
220	8.6.3 Broadcast	141
221	8.6.4 Virtualization	145
222	8.6.5 Break-before-make	148
223	8.6.6 Access permissions	150
224	8.7 Context synchronisation	153
225	8.7.1 Relaxed system registers	153
226	8.8 Problems	154
227	8.8.1 Reachability	155
228	8.8.2 Wide invalidations	155
229	8.9 Contributions	156
230	8.10 Related work	158
231	9 An axiomatic VMSA model	159
232	9.1 Candidate events	159
233	9.2 Candidate relations	162
234	9.3 Axioms	164
235	9.4 Relations	166

236	9.4.1	Observed-by	166
237	9.4.2	Dependency-ordered-before	166
238	9.4.3	Barrier-ordered-before	167
239	9.4.4	Translation-ordered-before	167
240	9.4.5	Contextually-ordered-before	168
241	9.4.6	Fault-ordered-before and ETS	169
242	9.4.7	TLBI-ordered-before	169
243	10	Validating the RVM model	174
244	10.1	Validation against the architecture	174
245	10.1.1	Clarity of architecture	174
246	10.1.2	Remaining questions and updates	174
247	10.2	Validating against hardware	175
248	10.2.1	Harness overview	175
249	10.2.2	Results from hardware	177
250	10.3	Validation by abstraction	178
251	III	Exceptions and interrupts	181
252	11	Relaxed precise exceptions	182
253	11.1	Introduction	183
254	11.1.1	Exception taxonomy	183
255	11.1.2	Exception lifecycle	184
256	11.1.3	Vectors and vector tables	184
257	11.1.4	Precision	185
258	11.2	Instruction instances	185
259	11.2.1	Fetch-decode-execute trees and streams	186
260	11.3	Relaxed behaviour of precise exceptions	187
261	11.3.1	Out-of-order execution across exception boundaries	187
262	11.3.2	Context synchronisation and speculation	188
263	11.3.3	Privilege level	188
264	11.3.4	Dependency through system registers	191
265	11.3.5	Ordering from asynchronous exceptions	191
266	11.3.6	Exception-specific mechanisms	191
267	11.3.7	Exceptions and the intra-instruction semantics	192
268	11.3.8	Disabling context synchronisation	192
269	11.4	Synchronous external aborts	193
270	11.4.1	Behaviour resulting from synchronous external aborts	193
271	12	An axiomatic model for precise exceptions	194
272	12.1	Extended candidates	194
273	12.2	Extended relations	196
274	12.3	Challenges in defining precision	196
275	12.4	Scope and limitations	197
276	13	Validating the exceptions model	198
277	13.1	Validating against hardware	198
278	13.2	Executable-as-a-test-oracle implementation	198
279	14	Conclusion	201
280	14.1	Limitations	201
281	14.2	Future work	202
282	A	Pocket guide to the Arm ISA	203
283	A.1	Architectural concepts	203
284	A.2	Guide to Instructions	205
285	A.2.1	Branches	205
286	A.2.2	Comparisons	208

287	A.2.3 Register moving and arithmetic	208
288	A.2.4 Memory accesses	210
289	A.2.5 Barriers	211
290	A.2.6 Cache maintenance	212
291	A.2.7 TLB maintenance	212
292	A.2.8 Exceptions	213
293	B The (i)Flat model	214
294	B.0.1 Intra-instruction Pseudocode Execution	216
295	B.0.2 Instruction Instance States	217
296	B.0.3 Thread States	218
297	B.0.4 Model Transitions	218
298	B.0.5 Auxiliary Definitions	223
299	B.0.6 Remarks about load/store exclusive instructions	223
300	C Test format: system-litmus-harness	225
301	D Proof of virtual memory abstraction	227
302	D.1 Abstraction	227
303	D.2 Anti-abstraction	227

Introduction

The computers we use every day are complex machines, made of many components, all working together to execute the software we run on them. These machines act as interpreters for a custom binary programming language, with commands made up of the instructions of the underlying *architecture*. These architectures can be thought of as abstractions of the hardware: programming languages whose syntax is defined by the binary encoding of the instructions from the *Instruction Set Architecture* (ISA), and whose semantics is the composition of the sequential behaviours of the instructions with a machine execution model. The architecture therefore can be thought of as the *interface* between hardware and software: defining the guarantees hardware must give and that software may rely upon.

Over the years much work has gone into defining, mathematically and precisely, the architectures that the processors we use every day implement. This previous work covers Intel/AMD's x86 [1, 2, 3, 4], Arm's ARMv7-A [5] and Armv8-A [6, 7] architectures, IBM's Power [8], RISC-V [9], and others. In theory, one might think that this interface would be straightforward to define. One can give precise formal semantics to the individual instructions, as Arm does, in its custom *Architecture Specification Language* (ASL) [10, 11], and then tie instructions together in a fetch-decode-execute loop. In practice, however, modern industrial architectures accumulate great complexity and subtlety. The Armv8-A and Intel reference manuals have 11,500 [12], and 4922 [1] pages respectively, covering everything from the individual instructions to the interactions between those instructions and the way they interact with memory.

The complexity of these interfaces becomes most apparent with the interaction with *multiprocessor* systems [13]. When multiple processors are executing concurrently, and communicating through shared memory, then various hardware optimisations, which are usually invisible to the programmer outside of timing effects, can become *architecturally visible*, affecting the semantics of the machine code, that is the values capable of being read or written to registers or memory by those processors. Over the years, these effects have been studied as part of the field of 'relaxed memory' research, resulting in numerous formal models for a variety of microprocessor architectures giving precise mathematical semantics to the concurrent behaviours of 'userland' machine code programs [14, 15, 3, 4, 16, 7, 17]. For high-level languages, there is similar work in understanding their relaxed memory behaviours which arise from both their compilation to such low-level machine programs and from the compiler's optimisations [18, 19, 20, 16].

We now seek to expand this work on relaxed memory for the Arm architecture, to cover not just those parts of the architectures used by userland processes, but the features required by systems software to function. In this work we will focus on the Armv8-A architecture: the *application*-class processors that power a large proportion of modern mobile devices. There are several reasons to focus on Arm: (1) they are ubiquitous, with over a trillion devices running Arm hardware, (2) Arm has a diverse ecosystem of implementations, meaning that software must program to this abstract interface much more tightly than one might for other architectures, and (3) Arm have put a large amount of effort into precisely and formally defining their ISA in their ASL language, enabling us to give a faithful specification to the architectural envelope.

Specifically, we will focus on key architectural features required by operating systems and hypervisors, which are not accessible, or only partially accessible, to userland processes: instruction fetching and cache maintenance, virtual memory and TLB maintenance, and exceptions.

1.1 Arm-A architecture overview

In this work we primarily focus on Arm. Arm will serve as an example representative modern microprocessor architecture, but many of the behaviours and conclusions will also apply to other architectures including RISC-V, IBM Power, and x86.

Arm produce three major classes of architectures, A-class (Application), R-class (Real-time) and M-class (Microprocessor). Arm principally make *architecture*. While Arm do design several implementations, many of which their partners use in their own designs, a number of partners also design their own implementations which they use in their own chips. This gives us a large surface of interesting designs to test, all implementing the same architecture. In particular, we will focus on the *Application* (A)-class processors.

Arm’s A-class architecture is intended to support general-purpose high-performance microprocessors, such as those found in mobile devices, tablets, laptops, and servers. Arm has three A-class architectures which can currently be found in modern hardware: ARMv7-A, Armv8-A, and Armv9-A. ARMv7-A is 32-bit only. Armv8-A and Armv9-A have both 32-bit and 64-bit modes, called *execution states*: AArch64 (for 64-bit execution) or AArch32 (for 32-bit execution). AArch64 mode uses the A64 instruction set. AArch32 mode can use either the T32 or A32 instruction sets. This is illustrated in Figure 1.1. We will focus here on the 64-bit architecture, AArch64 and its A64 ISA, as found in Armv8-A and Armv9-A. We use the term *Arm-A* to refer to both Armv8-A and Armv9-A interchangeably.

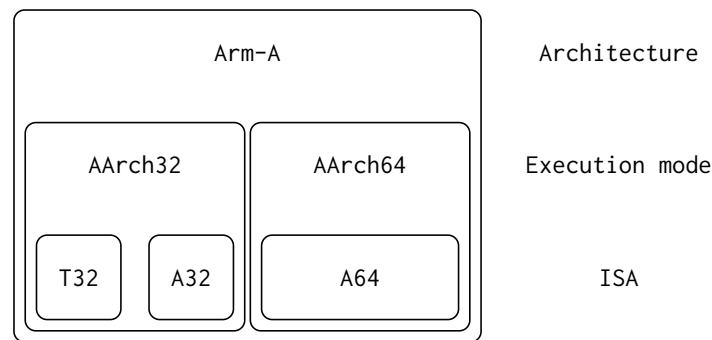


Figure 1.1: Arm-A structure.

At the time of writing, A64 has 402 ‘base’ instructions and another 1,205 vector, matrix and floating-point instructions. It has 31 general-purpose registers, accessible through either 32-bit views as $w0-w30$, or as 64-bit views as $x0-x30$, as shown in Figure 1.2. It has a dedicated zero register (wzr/xzr), and stack pointer register (sp). Instructions are fixed-width, with 32-bit opcodes, and in the typical RISC style: with most instructions reading operands from registers, and writing results back to registers, with only limited support for immediate values. Execution in AArch64 is split into 4 ‘exception levels’, these demark the levels of privilege that a process may have, ranging from EL0 (least privileged) to EL3 (most privileged). Typically *userland* processes execute at EL0, with very limited access to hardware features; with operating systems running at EL1, hypervisors running at EL2, and any firmware and secure monitor running at EL3. There are also secure modes, which we do not consider here. Each CPU has its own bank of registers; is executing in either AArch64 or AArch32 execution mode; is fetching, decoding and executing instructions from either the A64, A32 or T32 ISAs; and is executing at at one of EL0, EL1, EL2 or EL3.

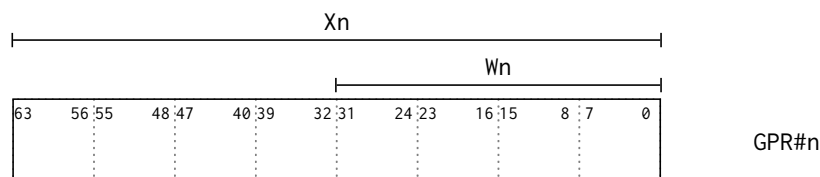


Figure 1.2: Arm-A W and X register views for a general-purpose register.

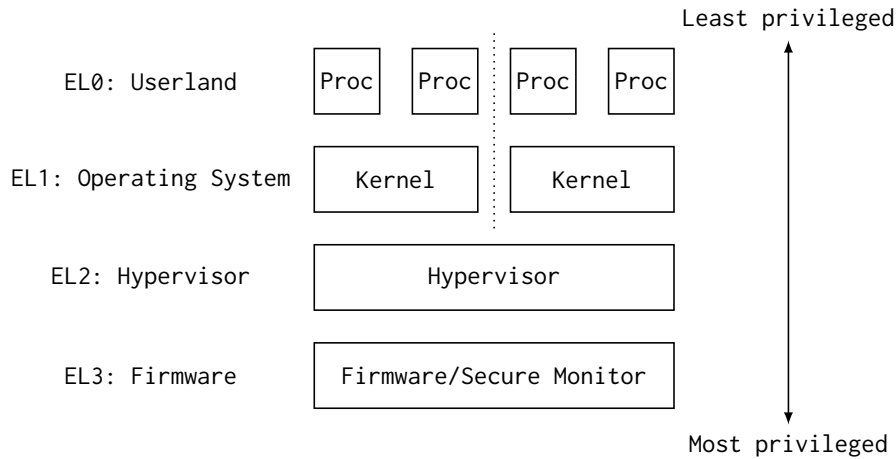


Figure 1.3: Arm-A exception levels.

1.2 Systems software

375

376 The programs we interact with on a day-to-day basis on our computers, our word processors and internet
 377 browsers, are typically unprivileged programs, with restricted access to hardware. Such programs are
 378 often referred to as executing in *userland*. These userland programs make up the bulk of the applications
 379 we use every day, from spreadsheets, to web browsers, text editors, and so on. They typically execute with
 380 the least privilege (in Arm, this means at EL0, as in Figure 1.3), and with the operating systems and
 381 hypervisors below them restricting the access to memory they have through the use of *virtual memory*
 382 (see Chapter 7).

383 Operating systems typically split userland execution into *processes*: discrete instances of programs, each
 384 with some associated dedicated (virtual) memory [21, p. 85]. It is then the operating system, executing
 385 with more privilege (at EL1), that configures and schedules these processes.

386 Modern operating systems seek to enforce isolation between these processes primarily through the
 387 application of a virtual memory abstraction [21, pp. 185,194,604][22, p 227] (described in detail in Part II),
 388 with each process behaving as if it had direct access to memory, when in fact the operating system (via
 389 the hardware supporting it) are redirecting the accesses at runtime.

390 This virtual memory abstraction can be layered, with an extra level of abstraction below the operating
 391 systems controlled by a *hypervisor*. Hypervisors behave similarly, but instead of controlling many processes
 392 at EL0 they instead can control multiple operating systems at EL1.

393 Finally, software at EL3 executes any firmware or secure monitor. Generally, the firmware performs
 394 hardware-specific actions, especially during boot (reading and writing implementation-defined configuration
 395 registers and performing any functionality required by the System-on-Chip). The Secure Monitor is a
 396 part of the Arm architecture's TrustZone security extensions, and we will not discuss these features here.

397 Figure 1.3 demonstrates a typical setup, with firmware running at EL3, a hypervisor at EL2, which is
 398 controlling a couple of operating systems, each of which has multiple processes under its control.

1.3 Relaxed memory

Hardware and software are not simply direct implementations of the sequential semantics one might expect of the languages they implement. Whether they are the machine languages implemented by microprocessors or software languages implemented by compilers, as time progressed these implementations have acquired multiple layers of abstraction, made with increasing complexity. Compilers and hardware optimise programs to be faster, use less space, and be more compact. They propagate and duplicate reads, subsume or outright eliminate writes, reorder operations in the program, replace one computation with another, or even just remove entire sections of the program entirely.

These optimisations may be *semantics preserving* with respect to the simple sequential semantics: aside from the timing effects they are designed to cause they are invisible to the programmer. This is, however, not true in all cases, with many highly desirable optimisations not preserving the source program's semantics [23].

It is multithreaded programs, and multicore processors, which often breaks the assumptions made by these optimisations. As an example, take Intel's x86 microprocessor architecture. It allows its implementations to perform an innocuous-sounding optimisation: to buffer writes together locally. This *store buffering* optimisation is ubiquitous in the hardware world, but it permits multiple cores to have mutually inconsistent views of memory [23, 3, 4]; where, at the same point in time, different cores see different values for the same memory address. If the programmer was unaware of these behaviours and the required mitigation in software, then this could break key invariants of software, leading to critical bugs in synchronisation primitives [23], data structures, or software more generally [24].

Intel, and their x86 architecture, is not the only example of hardware architectures performing such optimisations, and store buffering is not the only behaviour hardware exhibits. Arm [12], RISC-V [25], and IBM's Power [26] architectures all exhibit their own behaviours, with consequential requirements on software. Each of these microprocessor architectures comes with its own reference manual, with tens of thousands of pages attempting to describe these behaviours. These architectures are incomparable: the behaviours they allow are not subsets of one another. Instead, there are several optimisations that some architectures allow as observable behaviour, where others do not. Those optimisations include, but are not limited to, things such as: reordering of instructions, prefetching and caching of data and instructions, buffering of loads and stores, hierarchical cache layouts, and branch prediction with speculation down those branches. It is not that some implementations perform these optimisations while others do not, but that those architectures which allow such behaviours to be observed do not require that the hardware include relevant hazard checking or invalidations which would recover from 'bad' states.

It is not just hardware that has these concerns. A variety of software languages, including C and C++ [27, 28], Java [29, §17.4], Rust [30], and Haskell [31], have comparable behaviours derived both from similar optimisations done by their compilers and interpreters, but also inherited from the hardware they run upon.

Over the decades, the community has spent a large amount of effort in understanding the behaviours the hardware actually exhibits, by empirically observing what extant hardware does, by talking with architects and hardware designers about what they imagine hardware could do, now or in the future, and by building precise mathematical models which capture the architectural 'envelope' of allowable behaviours. These models come in many flavours, and in [Chapter 2](#) we will explore two such models for Arm, and the set of behaviours they are intended to capture.

1.4 Contributions

In this work, we extend the previous relaxed memory work on Arm into the realm of systems software: instruction fetch and cache maintenance (Part I), pagetables and TLB maintenance (Part II), and a start on exception handling (Part III). We produce a microarchitectural-style operational semantics for the first in the style of Flat [7], and herd-style declarative (‘axiomatic’) semantics [32] for all parts.

In Part I, we produce:

- ▷ A set of litmus tests (Ch.3) which cover a variety of phenomena and architectural features, including self-modifying code and the required cache maintenance, clarifying the architectural intent.
- ▷ A microarchitectural-style structural-operational-semantics (Ch.4) which implements that architectural intent as we understand it.
- ▷ A formulation as an axiomatic-style declarative semantics (Ch.5), intended equivalent to the aforementioned operational model, as an extension to the axiomatic model of [7].
- ▷ An extension of the `litmus7` tool [33], and a set of experimental results from testing against a range of hardware (Ch.6).

The models cover both the base architecture, as well as the implementation-defined choices for stronger cache types. The tests represent the architectural intent as we understand it, modulo mixed-size accesses and open questions where explicitly stated. The base model is thoroughly validated through discussions with architects and experimental testing of hardware, although at the time of the work the hardware did not support the stronger cache types. The two models are intended equivalent, and, although no formal proof-of-equivalence has been done, we have empirically tested their equivalence through the auto-generation of a large suite of tests with an extension to the `diy` [34] tool. Since publication of the work, Arm have extended the official Arm memory model with instruction fetching and cache maintenance, although we do not believe the architectural intent has changed since this work.

In Part II, we produce:

- ▷ A collection of litmus tests for virtual memory and TLB maintenance, for multi-level translation table walks with both stages (Ch.8), clarifying the architectural intent in many cases or identifying remaining open problems in others.
- ▷ An axiomatic-style declarative semantics (Ch.9), as an extension to the original Armv8 model.
- ▷ A new hardware testing harness, and validation of the models by experimentation against hardware, and through abstraction proofs (Ch.10).

We have a large collection of hand-written litmus tests, covering a substantial portion of the virtual memory systems architecture of Arm: translation table walks and the respective translation reads, caching in TLBs, TLB maintenance, virtualisation and multi-stage TLB invalidations, and how these all interact with the rest of the memory model. We further clarify the architectural intent in many of these areas, although in some places the architectural intent has further evolved since the original publication of the work; such places are explicitly stated as such, e.g. for forwarding and enhanced-translation-synchronisation. Arm has independent, but adjacent, work on virtual memory covering an overlapping set of features but with a focus on hardware support for access flags and dirty bits as required by KVM; we believe the architectural intent for the tests presented here has no changed, except where explicitly stated, and the models correspond on the tests.

In Part III, we produce:

- ▷ A set of litmus tests (Ch.11), covering precise exceptions, clarifying the architectural intent around precision and synchrony with respect to the weak memory model.
- ▷ An axiomatic-style declarative semantics for precise exceptions in Arm (Ch.12).
- ▷ An extension to the hardware testing harness of Chapter 10 to support hardware testing of exceptions, and experimental results from running tests on hardware (Ch.13).

We cover the base exception machinery, clarifying the ordering guarantees of precise exceptions, and exploring how external errors in the system can effect those guarantees. The model presented, to the best of our knowledge, captures this intent. We have validated the model against some hardware implementations, although much more is required for high confidence. We are aware of independent work by Arm in the same area at the time of writing, but do not know its status.

1.5 Publications and collaborations

The work presented in the three parts were done in collaboration with a variety of other people on different aspects, resulting in the following publications:

- ▷ ‘**ARMv8-A system semantics: instruction fetch in relaxed architectures**’, in the Proceedings of the 29th European Symposium on Programming (ESOP 2020), by **Ben Simner**, Shaked Flur, Christopher Pulte, Alasdair Armstrong, Jean Pichon-Pharabod, Luc Maranget, and Peter Sewell [35].
- ▷ ‘**Isla: Integrating full-scale ISA semantics, axiomatic concurrency models**’, in the Proceedings of the 33rd International Conference on Computer Aided Verification (CAV 2021), by Alasdair Armstrong, Brian Campbell, **Ben Simner**, Christopher Pulte, and Peter Sewell [36].
- ▷ ‘**Relaxed virtual memory in Armv8-A**’, in the Proceedings of the 31st European Symposium on Programming (ESOP 2022), by **Ben Simner**, Alasdair Armstrong, Jean Pichon-Pharabod, Christopher Pulte, Richard Grisenthwaite, and Peter Sewell [37].
- ▷ ‘**Isla: Integrating full-scale ISA semantics, axiomatic concurrency models (extended version)**’, in Formal Methods in System Design (May, 2023), by Alasdair Armstrong, Brian Campbell, **Ben Simner**, Christopher Pulte, and Peter Sewell [38].
- ▷ ‘**Precise exceptions in relaxed architectures**’, Accepted for publication at the 52nd International Symposium on Computer Architecture (ISCA 2025), arXiv pre-print, by **Ben Simner**, Alasdair Armstrong, Thomas Bauereiss, Brian Campbell, Ohad Kammar, Jean Pichon-Pharabod, and Peter Sewell [39].

Many of the aspects of the work presented in this thesis were done jointly with many of the people listed above. The Isla tooling was primarily written by Alasdair Armstrong. The work on the litmus and diy tools was done by Luc Maranget. The production of litmus tests and discussions with architects and microarchitects was done jointly with Shaked Flur, Christopher Pulte, Ohad Kammar, Thibaut Pérami, Jean-Pichon Pharabod, and Peter Sewell. The writing of models was done in collaboration with Christopher Pulte and Shaked Flur (for ifetch); Christopher Pulte and Thibaut Pérami (for VMSA); and Jean Pichon-Pharabod and Ohad Kammar (for exceptions). Validation of the models, through proof and hardware testing, was done jointly with Jean Pichon-Pharabod (on the VMSA abstraction proofs) and Luc Maranget (test generation and hardware testing for ifetch).

Much of the above work was done in collaboration with Arm and their staff, in particular their chief architect, Richard Grisenthwaite. He is our primary contact within Arm, and we have a close collaboration with him characterised by discussions on Arm hardware, the requirements of the software that runs on them, the consequences of the models we propose, and, where relevant, the history of the architecture. In cases where we present some behaviour and declare that it is ‘allowed by Arm’, it usually means we have confirmation from the chief architect directly. That said, the Arm architecture and its reference text is solely the responsibility of Arm, and the architectural intent can change over time. This thesis describes our current understanding at the time of writing; it is not an Arm publication. We also work closely with the creator of the original Armv8 memory model, Will Deacon, who maintained the official Arm model until 2020, and who is a co-maintainer of the Arm port of the Linux kernel. We also engaged in various discussions with many members of current and former Arm staff: Ian Caulfield, Nikos Nikoleris, Gustavo Petri, Anthony Fox, Nathan Chong, Martin Weidmann, Jade Alglave, and others, who discussed with us the models, hardware implementations, and provided feedback individually on many of the aforementioned publications.

1.6 Thesis overview

This document starts with an introduction to systems software and relaxed memory (Chapter 1), and a discussion of the technical details of the ‘usermode’ Arm model (Chapter 2). The remainder of the document is then split into three parts:

- ▷ Instruction fetch (Part I comprising chapters 3-6)
- ▷ Virtual memory (Part II comprising chapters 7-10)
- ▷ Exceptions (Part III, comprising chapters 11-13)

We end with a short conclusion (Chapter 14), which includes a discussion of the limitations of the work presented here and some reasonable next steps.

Background Chapter 2 covers the fundamental concepts behind relaxed memory: the idea of litmus testing as a means to clarify and understand architecture, including a selection of important and useful litmus tests from the literature; how Arm defines their intra-instruction semantics and how such semantics compose with a concurrency model; the two kinds of concurrency models we will explore in this thesis, microarchitectural-style operational semantics and axiomatic-style declarative semantics; and instantiations of these for Arm-A.

Part I: Instruction fetching We start with a brief overview of the existing prose text in the Arm-A specification for instruction fetch and the related instruction (and data) cache maintenance operations. Focusing primarily on self-modifying (and concurrent modification) of code, such as what is required for just-in-time compilers (JITs), dynamic loaders, and operating systems schedulers, we produce a set of litmus tests (Ch.3) to capture the key relaxed behaviours that arise from the optimisations found in modern microprocessors, and clarify where such behaviours were unclear. We produce a microarchitectural-style operational semantics (Ch.4) based on our discussions with architects and micro-architects. We give an axiomatic model (Ch.5) intended equivalent to the operational model. We then validate that these models (Ch.6), confirming they coincide for the litmus tests given in the chapter. We automatically generate a large test suite of novel tests and check the two models do not diverge on these tests, and additionally check that they do not forbid behaviours exhibited on hardware by running the test suite on a selection of modern Arm processors.

Part II: Virtual memory Structured similarly to the instruction-fetching chapters, but independently of them, we explore the Arm *Virtual Memory Systems Architecture* or *VMSA*. We begin with some background information in the form of an overview of the sequential aspects (Ch.7), describing the structure and behaviour of the Arm address translation and memory management architecture without considering concurrency or caching effects. Then, we explore the relaxed behaviours of virtual memory (Ch.8) by producing litmus tests and discussing the architectural intent. We produce an axiomatic-style model for relaxed virtual memory (Ch.9), as an extension to the original (user mode) model, using the whole Arm translation table walk, including multiple stages, and TLB maintenance. Finally, there is a discussion on the validation of this model (Ch.10) achieved by discussion with the Arm chief architect, along with some limited testing of current Arm hardware, and some proofs over the axiomatic model for some expected key abstraction results.

Part III: Exceptions We finish the trio with a short overview of the initial work on relaxed exceptions in Arm-A. We begin with a discussion on the Arm interpretation of precise exceptions and give litmus tests which capture the key phenomena (Ch.11); we then give an axiomatic model which capture those phenomena (Ch.12), and finally produce some preliminary hardware results to support the models (Ch.13).

Conclusion Finally, Chapter 14 presents a short summary of the presented work, its limitations, and relation to other work in the area. We discuss what was learned, in terms not only of the models produced but also of the process itself, before finally touching on what remains as potential future work.

Modelling Arm: background

Now we turn our attention to the now well-established methods of precisely and formally modelling relaxed memory behaviours, in the context of Arm-A. In this chapter, we will cover two methods: microarchitectural-style operational semantics, which mimic the mechanisms seen on hardware; and axiomatic-style declarative models which succinctly define the validity of whole-program executions.

The idea of *litmus testing* is central: litmus tests provide a way of succinctly and efficiently describing and enumerating the behaviours of the underlying architecture that the models should allow or forbid. We start by looking at litmus testing in general, and some specific litmus tests of interest to the Armv8-A models, before looking at the models in detail.

2.1 Relaxed behaviours and litmus testing

The foundation of much relaxed memory work has been focused on *litmus tests*, small, self-contained, executable, snippets of code. They each capture a simple pattern or shape one may find in software.

Take the classic MP (‘Message passing’) litmus test as an example [23]. The code listing for the AArch64 (Arm-A) variant can be found in Figure 2.1. The ‘MP’ portion of the name captures the *shape*: the core pattern (or precisely, the cycle) of events which act as the skeleton of the test. In this case, *message passing* is a common software pattern, where one thread writes some data followed by a flag signalling the data is ready, while another thread concurrently reads the flag in order to further read the data. Thus, the ‘MP’ shape implies a two-threaded test with two locations (typically named *x* and *y*), with one thread (typically written first) writing to the locations, and another thread reading them in the converse order. The second half of the name (‘+pos’) designates the *variation* on the shape, in this case, that both threads have accesses just program-order after each other with no other barriers or dependencies. Typically these variations are defined as the sequence of orderings between events (separated by - in the name) for each thread (separated by +). Thus, we get a whole *family* of litmus tests based on the basic MP shape: MP+pos (the one shown here), MP+dmbs (with an Arm `dmb` memory barrier on each thread), MP+dmb.st+addr (with an Arm `dmb.st` memory barrier on the writer thread and an address dependency on the reader thread), and so on.

MP+pos		AArch64
Initial state: 0:X1=x, 0:X3=y, 1:X1=y, 1:X3=x, *x=0, *y=0		
Thread 0	Thread 1	
MOV X0,#1 STR X0,[X1] MOV X2,#1 STR X2,[X3]	LDR X0,[X1] LDR X2,[X3]	
Allowed: 1:X0=1, 1:X2=0		

Figure 2.1: MP test code listing.

607 The code listing given is totally standard [40]: the top line contains the name of the litmus test (MP+pos),
 608 and the architecture that this variant is for (AArch64); the second section contains the initial register
 609 and memory state; the next section contains the assembly code listing for each thread; and finally at the
 610 bottom is a conjectured outcome (plus its architectural intent, if known) given as a constraint on the final
 611 register and memory state. On Arm, the outcome given in the listing in Figure 2.1 is allowed.

612 On a sequentially consistent (*SC*) machine, whose executions are simply the interleaving of the instructions
 613 of all threads [41], there are many executions of the listed code, each giving rise to (potentially distinct)
 614 final states. To see the highlighted outcome, where Thread 1 reads 1 for *y* but 0 for *x*, there is only one
 615 possible combination of reads: that the read of *y* reads from the write to *y*, and the read of *x* reads from
 616 the initial memory state. This combination is not consistent with any of the simple interleavings of the
 617 instructions that a sequentially consistent machine would perform. We represent these executions not
 618 as an interleaving of the instructions, but as a graph of the events of those instructions (the reads and
 619 writes they perform) connected by their implicit orderings. There may be, and in this case, are, multiple
 620 different operational traces that lead to the same execution witness, which we shall explore later. The
 621 execution graph that corresponds to the allowed outcome can be found in Figure 2.2.

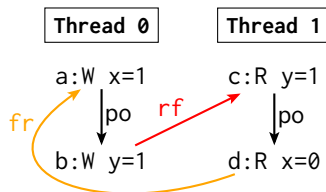


Figure 2.2: MP test execution diagram.

622 The nodes on the left, below the Thread 0 label, correspond to events from executing Thread 0 of the
 623 program. The event labelled *a* corresponds to the propagation of the first store in Thread 0 (the write
 624 of 1 to *x*) to memory, and event *b* corresponds to the write of the second store being propagated. They
 625 are related by program-order (*po*), indicating that the instruction *a* came from is earlier than that of *b*
 626 in the instruction stream of the processor; that is, *a* comes before *b* in the control flow of that thread,
 627 as determined by the order the processor fetched and decoded the instructions in. Similarly, below the
 628 Thread 1 label we see the event labelled *c*: the read event corresponding to the first load, reading the
 629 address *y* and getting the value 1. The value read came from the write event *b*, therefore *b* is related to *c*
 630 by the reads-from (*rf*) relation. Finally, the load of *x* reads from the initial value in memory, so we have
 631 another read event, labelled *d*, which reads 0. The read *d* of *x* read a value from a write to *x* from before
 632 the event *a* happened. In this case, that write is the initial write from the ‘Initial state’ of the test, and so
 633 *d* is related to *a* by the from-reads (*fr*) relation.

634 On Arm, the writes and reads need not execute in the order they appear in the program. So, while this
 635 execution appears to have a cyclic dependency in the order events must have happened in, the cycle can
 636 be broken by re-ordering the execution of either the reads or writes. The execution is therefore allowed,
 637 and we readily observe this outcome on most modern hardware.

638 **Litmus testing** We use litmus tests to explore *behaviours*: particular patterns in code, or specific
639 hardware mechanisms that are responsible for allowing or forbidding the test. Many litmus tests exercise
640 many microarchitectural mechanisms whose composition or confluence leads to the final result, or where
641 there may be multiple different mechanisms or choices that could each independently lead to the same
642 result. For example, in the MP+pos test we just saw, there are three well-understood microarchitectural
643 explanations: that the stores are committed out-of-order (re-ordered within the pipeline, store queue, or
644 other thread-local storage), that the stores propagate out-of-order (are pushed out-of-order into the shared
645 memory), or that the loads satisfy out-of-order (either requested out-of-order in the pipeline, or requests
646 returned out-of-order from the memory subsystem). Any of the above explanations are alone sufficient to
647 allow the relaxed outcome highlighted by the test. One needs to prevent out-of-order execution on both
648 sides of the test (through the use of memory barriers, for example) to forbid that relaxed outcome.

649 Previous work has systematically enumerated these various patterns to produce a large collection of litmus
650 tests, for a range of architectures, each with an assortment of variations for different intra-thread orderings
651 (for barriers, dependencies, and so on). This has included obtaining both the architectural intent for
652 those patterns, as well as extensive testing campaigns on a variety of modern hardware. In some cases,
653 some outcome may be *architecturally allowed*, that is, the final state constraint is permitted to occur in
654 practice, but has not been experimentally observed on any hardware so far. In other cases, there may
655 be no architecturally allowed execution that permits a particular outcome, but it is still observed on
656 hardware: these are (or at least imply there exist) *hardware errata*, more commonly referred to as ‘bugs’.
657 We will not do an exhaustive review of all the behaviours that are allowed and forbidden in Arm, instead
658 referring the reader to the existing literature [14, 40, 32, 16, 7, 6, 42]. However, we will briefly look at
659 some of the behaviours that the reader should be familiar with in order to understand future chapters,
660 namely coherence, barriers and dependencies, and multi-copy atomicity.

661 2.1.1 Thread-local ordering

662 On Arm, instructions need not execute in the order they appear in the program, as we just saw. Reads
663 and writes are free to be re-ordered with respect to each other, with few restrictions. This is in contrast
664 to other architectures such as Intel/AMD’s x86, where only writes can be re-ordered with respect to
665 program-order later reads (through store buffering) [1, 23, 3]. Note that this does not mean that the
666 hardware is not allowed to re-order the instructions, but that if it does, it must preserve the illusion of
667 in-order execution to the programmer.

668 Not all re-orderings are permissible. In particular, Arm requires that single-threaded programs should
669 behave as if executed sequentially, at least for loads and stores. This means that non-SC executions only
670 come about through the interaction between multiple threads. We have already seen this with the MP test
671 earlier. To forbid the outcome of that test we must add barriers or dependencies to enforce thread-local
672 ordering, preventing the events from being reordered. Two (forbidden) variations of MP can be found in
673 Figure 2.3.

674 Dependencies in Arm arise from the intrinsic control and data flow of the program. Usually, they are
675 categorised into three kinds: address dependencies (*addr*), from reads to memory events that use that
676 read in the computation of the address the memory event accesses; data dependencies (*data*), from reads
677 to writes, where the value read is used in the computation of the value written; and control dependencies
678 (*ctrl*), from reads to events of instructions program-order after a (conditional) branch in the program
679 where the value of the read was used in the computation of the value used in the condition. Note that these
680 are not purely dynamic properties of the execution, but rather they are *syntactic* in that the dependencies
681 an instruction induces is a statically known property, thus there are no so-called ‘fake’ dependencies: the
682 values read or written at runtime by an instruction does not matter; only the set of registers it accesses.

683 Not all dependencies are equal. On Arm, address and data dependencies enforce both read-to-read
684 and read-to-write ordering, whereas control dependencies enforce read-to-write but not read-to-read
685 ordering. Speculation allows reads to happen ‘early’, but not writes; this gives an asymmetry where
686 control dependencies provide strength to a write but not a read. This can be seen in the two tests in
687 Figure 2.4.

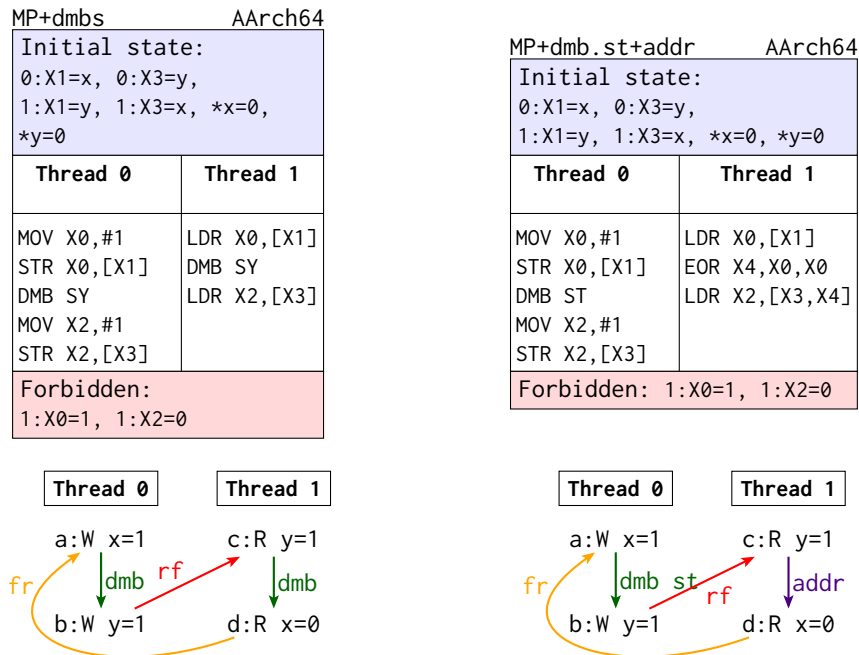


Figure 2.3: Two variants of MP with thread-local ordering.
 On the left: MP+dmb with Arm DMB barrier between instructions.
 On the right: MP+dmb.st+addr with an address dependency between the reads.

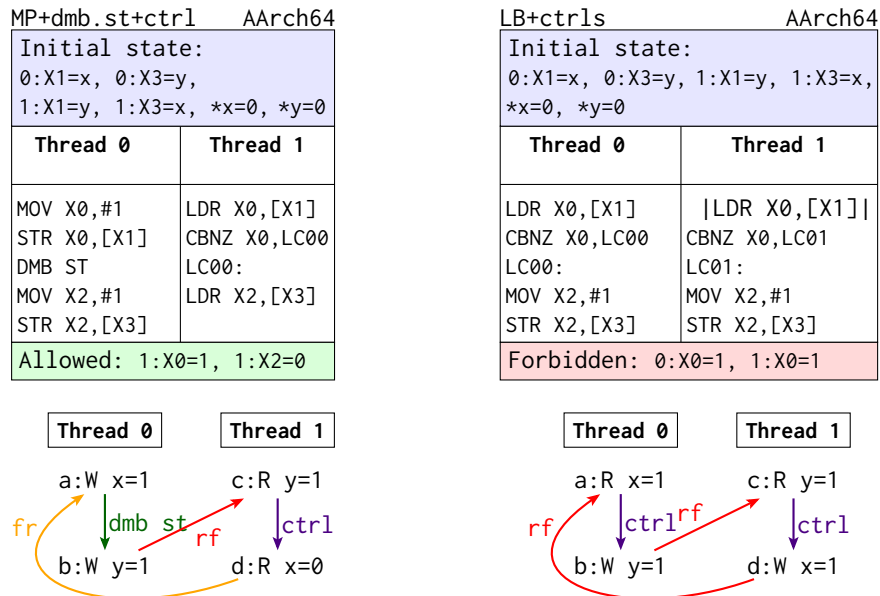


Figure 2.4: Two litmus tests with speculation.
 On the left: MP+dmb.st+ctrl with Arm DMB barrier between the writes, but a control dependency between the reads.
 On the right: LB+ctrls, a variant of the classic ‘load buffering’ litmus test, with control dependencies to both writes.

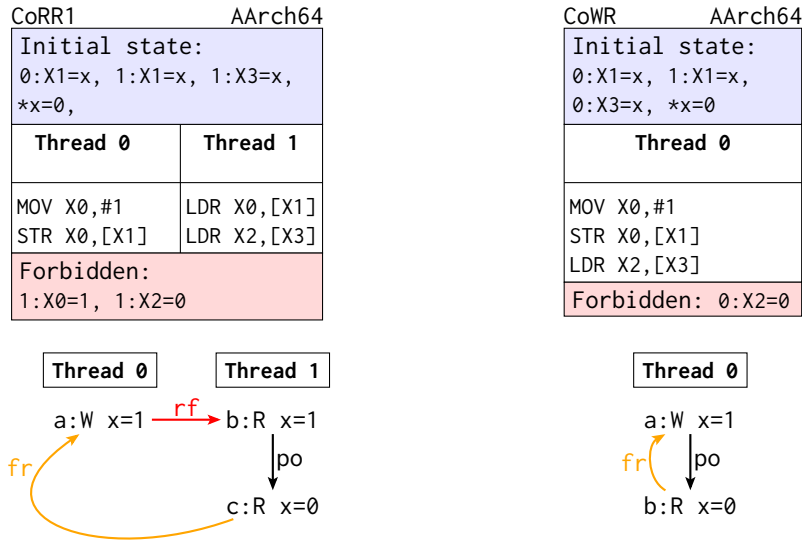


Figure 2.5: Two coherence litmus tests. On the left: CoRR1, that two subsequent reads of the same location in the same thread should be consistent with the coherence order. On the right: CoWR, that a read of a location cannot skip over a newer program-order earlier write from the same thread.

688 2.1.2 Coherence

689 A guarantee provided by most modern microprocessor architectures is *coherence*: that there is, for each
690 location, a total order that writes to that location happen in, that all threads agree on [8]. Microarchi-
691 tecturally, this emerges naturally from the desire to ensure that writes are never dropped by the cache
692 protocol, and since writes may be of sub-cache-line size (down to individual bytes) the cache protocol
693 must ensure consistency over whole cache lines at a time.

694 This property is one that sets processor consistency models apart from those one would find in databases
695 and other distributed systems, which generally do not require it, such as the classic *causal consistency*
696 model for distributed systems [43].

697 Two of the key litmus tests for coherence can be found in Figure 2.5.

698 2.1.3 Multi-copy atomicity

699 Coherence is not sufficient to guarantee that all threads agree on what the most recent write is at the
700 same point in time.

701 In particular, while all threads will see the same writes to the same location in the same order, at any
702 particular moment some threads may not have caught up to the latest write yet. Architectures that have
703 this property are called *non-multi-copy atomic* [13].

704 Arm has a kind of partial multi-copy atomicity, which they term *other-multi-copy atomicity*. Other-multi-
705 copy atomicity gives guarantees similar to multi-copy-atomicity, but allows writes to be read by the
706 writing thread itself earlier than they can be seen by other threads, however, once a write has propagated
707 to another thread then all threads must see that write or something newer [7]. The hardware mechanism
708 which motivates this is *write forwarding*: the processor can satisfy a read from a same-thread same-location
709 program-order-earlier write, if that write has committed, even before the write has propagated out to
710 memory. Figure 2.6 contains the classic PPOCA (preserved-program-order-control-address) litmus test,
711 which shows that writes can be observed locally before being propagated to other threads, even down
712 speculative branches. Figure 2.7 shows the IRIW (independent-reads independent-writes) litmus test,
713 which demonstrates the latter point, that writes propagate to all threads simultaneously.

MP+dmb.st+addr-rfi-addrch64

Initial state:	
0:X1=x, 0:X3=y, 1:X1=y, 1:X3=x, 1:X3=z, 1:X5=z, *x=0, *y=0, *z=0	
Thread 0	Thread 1
MOV X0,#1 STR X0,[X1] DMB ST MOV X2,#1 STR X2,[X3]	LDR X0,[X1] EOR X8,X0,X0 MOV X2,#1 STR X2,[X3,X8] LDR X4,[X5] EOR X9,X4,X4 LDR X6,[X7,X9]
Allowed: 1:X0=1, 1:X4=1, 1:X6=0	

PPOCA AArch64

Initial state:	
0:X1=x, 0:X3=y, 1:X1=y, 1:X3=z, 1:X5=z 1:X8=x, *x=0, *y=0	
Thread 0	Thread 1
MOV X0,#1 STR X0,[X1] MOV X2,#1 STR X2,[X3]	LDR X0,[X1] CBNZ X0,LC00 LC00: MOV X2,#1 STR X2,[X3] LDR X4,[X5] EOR X6,X4,X4 LDR X7,[X8]
Allowed: 1:X0=1, 1:X4=1 1:X7=0	

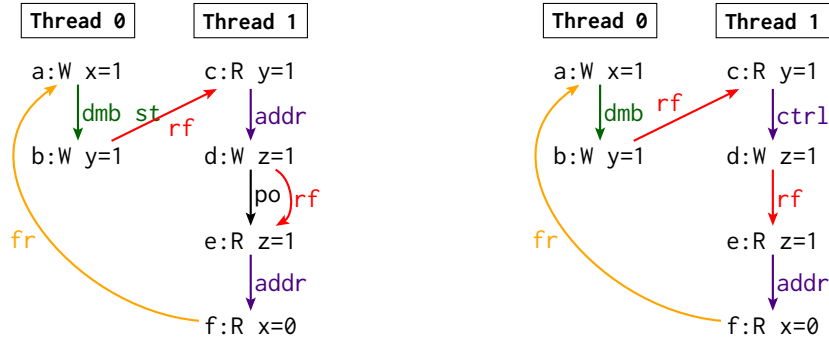


Figure 2.6: Two litmus tests with write forwarding. On the left: MP+dmb.st+addr-rfi-addr with write-forwarding down a non-speculative branch. On the right: PPOCA, with write-forwarding down a speculative branch.

IRIW+dmbs AArch64

Initial state: 0:X1=x, 1:X1=x, 1:X3=y, 2:X1=y, 3:X1=y, 3:X3=x, *x=0, *y=0			
Thread 0	Thread 1	Thread 2	Thread 3
MOV X0,#1 STR X0,[X1]	LDR X0,[X1] MOV X2,#1 DMB SY LDR X2,[X3]	MOV X0,#1 STR X0,[X1]	LDR X0,[X1] MOV X2,#1 DMB SY LDR X2,[X3]
Forbidden: 1:X0=1, 1:X2=0, 3:X0=1, 3:X2=0			

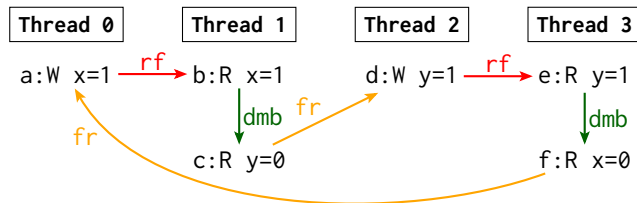


Figure 2.7: IRIW+dmbs: a classic multi-copy atomicity litmus test.

2.2 Intra-instruction semantics

Previous work has, for Arm and RISC-V, established high-fidelity models for the *intra*-instruction behaviour of individual instructions. That is, the sequential behaviour of the register and memory accesses, and any arithmetic over them, the instruction performs.

Arm produces such models as part of their architecture specifications, in their custom ASL (*architecture specification language*) programming language [10], which can be found in the manual [12] or otherwise acquired from Arm [44].

The ASL and Sail specification languages Although this work is focused on Arm-A and Arm use ASL, the tools we build upon are architecture-agnostic and use the Sail specification language for instruction semantics [45]. We therefore use the automatically generated ASL-to-Sail translation [45, 46] of the official Arm specification for most of the examples presented here, except where specified otherwise. Sail and ASL are very similar languages, and are used for broadly the same purposes, with similar syntax and semantics; we will not go into depth here into the history or minutiae of them; instead, we will look at just one aspect of Sail, its primitive effects, as it is important to the function of the tools we will use later on.

Outcomes Sail programs have effects for the interaction between the instruction semantics and the registers and memory. These effects make Sail programs monadic computations over the Sail effect datatype (called outcome). Figure 2.8 lists the outcomes defined by the Sail standard library [15], it contains one pure value (DONE), and the other values each represent one step of the intra-instruction semantics suspended at the interface with the environment, containing a continuation to resume the execution with the environment's choice.

READ_MEM(read_kind, address, size, read_continuation)	Read request
WRITE_EA(write_kind, address, size, next_state)	Write effective address
WRITE_MEMV(memory_value, write_continuation)	Write value
BARRIER(barrier_kind, next_state)	Barrier
READ_REG(reg_name, read_continuation)	Register read request
WRITE_REG(reg_name, register_value, next_state)	Write register
INTERNAL(next_state)	Pseudocode internal step
DONE	End of pseudocode

Figure 2.8: Outcomes (the Sail effect datatype).

An example instruction As an example, take the Arm ‘ADD Xd,Xn,Xm’ instruction, whose Sail code can be found in Figure 2.10, as translated from the original source ASL code in the Arm manual. It takes two input registers (Xn,Xm), adds the values stored in them together, and stores the result in the output register (Xd), updating any flags as it does so.

The calls to `x_read` and `x_set`, and (not shown) `EndOfInstruction`, each generates an effect. Omitting the outcomes for the flag registers, and the exact arithmetic calculation, executing this code results in the trace of outcomes shown in Figure 2.9:

```
Read_reg(n, fun v1 ->
  Read_reg(m, fun v2 ->
    Write_reg(d, (v1 + v2), Done)
  )
)
```

Figure 2.9: Trace from the Arm ADD instruction.

The set of such traces define the semantics of that instruction, and the concurrency models described later in this chapter are parameterised over such traces.

```

1  function execute_aarch64_instrs_integer_arithmetic_add_sub_shiftedreg (d,
   datasize, m, n, setflags, shift_amount, shift_type, sub_op) = {
2     result : bits('datasize) = undefined;
3     let operand1 : bits('datasize) = X_read(datasize, n);
4     operand2 : bits('datasize) = ShiftReg(datasize, m, shift_type, shift_amount)
   ;
5     nzcvc : bits(4) = undefined;
6     carry_in : bits(1) = undefined;
7     if sub_op then {
8         operand2 = not_vec(operand2);
9         carry_in = 0b1
10    } else {
11        carry_in = 0b0
12    };
13    (result, nzcvc) = AddWithCarry(operand1, operand2, carry_in);
14    if setflags then {
15        (PSTATE.N @ PSTATE.Z @ PSTATE.C @ PSTATE.V) = nzcvc
16    };
17    X_set(datasize, d) = result
18 }

```

Figure 2.10: Sail pseudocode for the ADD Xd,Xn,Xm instruction.

2.3 Arm-A operational model

The state of the art multi-copy-atomic operational semantics for Arm is the *Flat* model [7, 6, 47] by Flur, Pulte, et al. Flat is a small-step operational semantics, with transitions designed to (abstractly) match the kinds of actions we see in hardware. It is implemented as an executable-as-a-test-oracle model in the RMEM tool [48].

RMEM is written in a combination of OCaml, and the Lem [49, 50] language for operational semantics. It can either be run through a command-line interface, for example to run batches of tests, or can be used interactively, including through a version compiled to JavaScript which can be run in a web browser [51].

Flat has an explicit flat memory (from which it derives its name), which stores the most recent write that propagated to memory for each location, and a set of hardware threads, with each thread containing a tree of concurrently executing instruction instances with explicit out-of-order execution (abstractly modelling modern microprocessor pipelines).

Figure 2.11 illustrates a snapshot of an example instruction tree from a thread with 10 in-flight instruction instances. Some instructions (i_2 , filled-in blue) have finished executing, some (i_3, i_6, i_7, i_9 , blank/white) have not begun executing, and some (i_0, i_1, i_4, i_8, i_5 , partly-filled blue) are currently in-progress. Flat has explicit speculation down branches, and re-ordering of instructions. This can be seen in the diagram: there is a fork in the tree at i_3 (a branch in the program) which has not yet been executed while some earlier instructions (i_0, i_1) have not finished (and so it is not yet known whether the program will execute down branch i_4 or i_8), but later instructions down both branches have already begun executing.

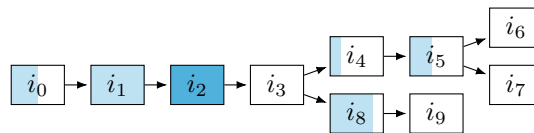


Figure 2.11: A tree of 10 concurrently executing instruction instances.

Flat is therefore composed of two subsystems: the thread subsystem which contains the pool of threads with their instruction trees, a storage subsystem which contains a flat array for memory, as sketched in Figure 2.12.

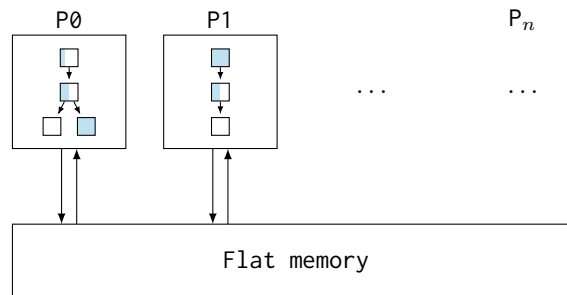


Figure 2.12: Flat state (diagram).

Thread subsystem More precisely, the thread subsystem has a per-thread tree of instruction instances. Each node in the tree is an instruction *instance*, a piece of state representing a single instruction in the process of being fetched, decoded and executed; its state includes the current pseudocode state (such states are listed in Figure 2.13), as well as any other ancillary data required by the operational model (pending addresses and values and so on).

The thread system then has a set of guarded transitions, split into two groups: the local transitions, each of which calls the continuation contained within the outcome of an instance and updates the instruction instance state with the new outcome; and, the synchronised transitions which can also update the storage subsystem state, which typically update the current pseudocode state without calling the continuation.

774 Figure 2.14 contains a fragment of the Lem code from RMEM which defines the thread subsystem state
 775 and the relevant transitions (but not their guards).

PLAIN(next_state)	Ready to make a pseudocode step
PENDING_MEM_READS(read_cont)	Performing the read(s) from memory of a load
PENDING_MEM_WRITES(write_cont)	Performing the write(s) to memory of a store

Figure 2.13: Operational pseudocode states.

```

1  type threadSubsystem =
2    nat → instruction_tree;
3  type instruction_tree =
4    list (instruction_instance *
5          instruction_tree);
6  type instruction_instance =
7    <| id: nat;
8      program_loc: address;
9      micro_op_state: micro_op_state;
10     mem_reads: set address;
11     ... |>
12 type micro_op_state =
13   | MOS_plain
14   | MOS_pending_mem_read
15     of outcome
16     of (value → outcome)
17   | MOS_potential_mem_write
18     of outcome
19 type thread_trans =
20   | T_register_read
21     of reg_name * value
22     of reg_name * value
23   | T_satisfy_read
24     of value
25   | T_mem_write_footprint
26     of list write
27   | T_mem_potential_write
28     of list write
29   | T_commit_store
30   | T_complete_store
31   | T_commit_barrier
32     of barrier_kind
33   | ...
34 type sync_trans =
35   | T_propagate_write
36     of write
37   | T_satisfy_read
38     of read_request * value
39   | T_propagate_barrier
40     of barrier_kind
41   | ...

```

Figure 2.14: Lem fragment of thread subsystem state.

776 **Storage subsystem** The Flat storage subsystem is comparatively straightforward: a finite map from
 777 location to the most-recently propagated write to that location. Figure 2.15 contains a fragment of the
 778 Lem sources from RMEM for the (non-mixed-size) Flat storage subsystem.

```

type flat_storage_subsystem_state = <| memory: nat → write; ... |>

```

Figure 2.15: Simplified Lem listing of the Flat storage subsystem state from RMEM.

779 **Transitions** Flat defines a set of common transitions for all instructions, as well as a set of specific
 780 transitions for stores, loads, and barriers. Below is a complete list of the local and synchronised transitions.

Common transitions

- ▷Fetch instruction
- ▷Pseudocode internal step
- ▷Register read
- ▷Register write
- ▷Finish instruction

Transitions on a Store instruction

- ▷Initiate memory writes of store instruction, with their footprints
- ▷Instantiate memory write values of store instruction
- ▷Commit store instruction
- ▷Propagate memory write
- ▷Complete store instruction (when its writes are all propagated)

Transitions on a Barrier

- ▷Commit barrier

Transitions on a Load instruction

- ▷Initiate memory reads of load instruction
- ▷Satisfy memory read by forwarding from writes
- ▷Satisfy memory read from memory
- ▷Complete load instruction (when all its reads are entirely satisfied)

782 Each transition has a guard, a predicate over the state that must be true in order for the transition to be
 783 valid, and an action, a function that updates the whole system state from one configuration to another.
 784 See Appendix B for a full rendering of the Flat model.

2.4 Arm-A axiomatic model

In contrast to the operational model presented in the previous section, a model with equivalent behaviour can be given declaratively, in a so-called *axiomatic* style. Axiomatic models describe the allowed behaviour of programs by a predicate, typically described by a collection of axioms, constraining the event of the *candidate executions* of that program. These candidates are the graphs of events of a single run of the program, with the events related by a set of intrinsic relations capturing the order of events and their dependencies.

The model first considers an overapproximate set of such candidates: executions consistent with the intra-instruction semantics, but where the values used in the program are unconstrained. The model then has axioms, generally acyclicity or emptiness of relations over the events, which reject some of these executions as *inconsistent*. Those that remain are the *valid*, or *consistent*, executions of the program.

The model can therefore be used to assert whether some given program can reach a final state satisfying some constraint. If there is a candidate executions of the program, which is consistent with the axioms of the model, then the model is said to *allow* that execution, and if the final state satisfies the given constraint, that outcome is permitted by the model.

Succinctly, an axiomatic model winnows down a large set of graphs of potential whole-program executions to a small set of allowed executions by checking that the events of those executions do not violate any of the axioms of the model.

2.4.1 Arm-A candidate executions

Arm-A candidate executions are composed of two parts: the set of events of the program, which for Arm these are the memory access and barrier events, labelled with their access type (read or write, or barrier kind); and the candidate relations over those events, such as program-order and address/control/data dependencies, amongst others.

It is often useful to split the candidate execution definition into two steps: first, to define a *pre-execution* which contains all the events, and the relations which are intrinsic to the program; then to complete these into a *candidate* execution with existentially-quantified relations coherence-order and reads-from, which witness a particular choice of runtime execution order.

More formally, we can define an Arm-A candidate execution as: a set of event IDs (here just assuming IDs are the natural numbers); a labelling function (from \mathbb{N} to Label); a collection of the candidate relations (\mathcal{C}_R) satisfying some constraints (described in more detail later on), and a candidate witness (\mathcal{C}_W) describing the existentially quantified coherence-order and reads-from relations.

$$\begin{aligned} \text{Candidate Pre-Execution} &\equiv \mathcal{P}(\mathbb{N}) \times (\mathbb{N} \rightarrow \text{Label}) \times \mathcal{C}_R \\ \text{Candidate Execution} &\equiv \text{Pre-Execution} \times \mathcal{C}_W \end{aligned}$$

The candidate relations, and the candidate witness, are sets of named relations over the events of the pre-execution, subject to some well-formedness constraints (discussed later):

$$\begin{aligned} \underline{L} &\equiv \mathbb{N} \times \mathbb{N} \\ \mathcal{C}_R &\equiv \langle \underline{po}, \underline{loc}, \underline{addr}, \underline{ctrl}, \underline{data}, \underline{rmw}, \underline{ext} \rangle \\ \mathcal{C}_W &\equiv \langle \underline{co}, \underline{rf} \rangle \end{aligned}$$

812 **Events** The labelling function maps each event ID to an event label, describing the kind of access and, if
 813 applicable, what data or address it operates over.

814 A simplified version of the labels, sufficient for the model described here, contains:

- 815 1. memory events with location and values, namely reads (R) including acquire reads (A) and weak-
 816 acquire reads (Q), writes (W) including release writes (L); and
- 817 2. a set of Arm barriers (DMB, ISB) and their variants.

More precisely, these labels can be described as follows:

$$\begin{aligned}
 \text{Label} &\equiv \text{Reads} \cup \text{Writes} \cup \text{Barriers} \\
 \text{Reads} &\equiv \{\mathbf{R}, \mathbf{A}, \mathbf{Q}\} \times \text{Loc} \times \text{Val} \\
 \text{Writes} &\equiv \{\mathbf{W}, \mathbf{L}\} \times \text{Loc} \times \text{Val} \\
 \text{Barriers} &\equiv \{\mathbf{DMB.LD}, \mathbf{DMB.ST}, \mathbf{DMB.SY}, \mathbf{ISB}\} \\
 \text{Loc} &\equiv \text{Bitvec}_{48} \\
 \text{Val} &\equiv \text{Bitvec}_{64}
 \end{aligned}$$

818 In §2.5.1 we will see a more realistic definition of the event types for a production architecture (Armv9-A),
 819 and their correspondence to the underlying effects of the Sail definition, as used by `isla-axiomatic`.

820 **Candidate relations** The candidate relations capture the relationships and orderings between the events
 821 of the execution. For Arm, the relations in a pre-execution are, with their intended meaning:

- 822 ▷ program order: $E_1 \text{ po } E_2$ iff the instruction generating E_1 occurs before the instruction generating
 823 E_2 in the instruction stream.
- 824 ▷ same-location: $M_1 \text{ loc } M_2$ iff the address of M_1 is the same location as used by M_2 .
- 825 ▷ address dependent: $R_1 \text{ addr } M_2$ iff the value read by R_1 is used in the calculation of the address
 826 M_2 .
- 827 ▷ data dependent: $R_1 \text{ data } W_2$ iff the value read by R_1 is used in the calculation of the value written
 828 by W_2 .
- 829 ▷ control dependent: $R_1 \text{ ctrl } E_2$ iff the value read by R_1 is used to determine whether or not the
 830 instruction E_2 originates from would have executed at all.
- 831 ▷ read-modify-write: $R_1 \text{ rmw } W_2$ for the separate read and write events of an atomic update.
- 832 ▷ external: $E_1 \text{ ext } E_2$ iff the instructions which generated events E_1 and E_2 originated from different
 833 hardware threads.

834 Plus the existentially quantified witness:

- 835 ▷ reads-from (rf), from W_1 to R_2 when R_2 reads the value that W_1 wrote.
- 836 ▷ coherence-order (co), from W_1 to W_2 where W_1 appears before W_2 in the coherence order of that
 837 location, (informally, that W_1 propagated to memory before W_2).

838 where R_n is a read event, W_n is a write event, M_n is a read or write, and E_n an event of any type.

839 **Implicit events and ISA-Consistency** Candidate executions for Arm contain a limited set of events:
 840 reads, writes, and barriers. In reality, there are many more *implicit* events, there in principle but typically
 841 excluded from the candidates: register reads and writes, instruction fetches, translation table walks, and
 842 so on. We will (in Parts I, II and III) extend the candidate executions to include some of these implicit
 843 events. For a candidate execution to be consistent with a given architecture’s intra-instruction semantics,
 844 as defined by its ISA, there must be a corresponding execution of that intra-instruction semantics, with
 845 all its implicit events, which corresponds to the set of explicit events seen in the candidate.

846 It is uncommon to include such events in the axioms and relations directly, and therefore are typically
 847 elided entirely. However, we could imagine that for each candidate execution, one can ‘complete’ the
 848 events to include all the hidden implicit events, such that the events of the candidate correspond exactly
 849 to the Outcome type of the intra-instruction semantics (see Figure 2.8).

850 Let us say that an execution E' is a completion of E , for which we will write $\text{Completion}(E', E)$ if the
 851 subgraph of E' restricted to explicit events is equal to E , and where the events within an instruction are
 852 ordered by an intra-instruction-causality order (iico) consistent with the definition of address/control/data

853 dependencies, as in Alglave et al. [14]. We can now define what it means for an execution to be consistent
 854 with an ISA given some intra-instruction semantics: that, roughly, there are a set of implicit events which
 855 complete the candidate such that the groups of events for each instruction correspond to traces from the
 856 intra-instruction semantics.

Formally, we can imagine partitioning the completed execution with its implicit events by program-order,
 to get groups of events corresponding to instructions, with traces of events given by `iico`. Let us call the
 set of `iico`-traces of an execution $\text{IICOTraces}(E)$. We can then define ISA-consistency by asking whether
 there exist traces in the intra-instruction semantics which correspond to the `iico`-traces of instructions in
 the execution:

$$\begin{aligned} \text{SimulatesISA}(E : \text{Execution}) : \forall t. t \in \text{IICOTraces}(E) \Rightarrow t \in \text{ISA} \\ \text{ISA-CONSISTENT}(E) = \exists E'. \text{Completion}(E', E) \wedge \text{SimulatesISA}(E') \end{aligned}$$

857 In practice, we generally go the other way: tools produce complete traces from the intra-instruction
 858 semantics defined by the ISA, and then discard or hide some events to obtain a smaller candidate —
 859 thereby producing ISA-Consistent executions by construction.

860 As an example, take the reader thread of an MP-shaped test, with a barrier between the loads. Figure 2.16
 861 shows a sketch for a completion of that reader thread for Arm, including general-purpose register and
 862 instruction fetch events (but still eliding the voluminous configuration register access and translation table
 863 walk events for brevity) with introduced implicit events in blue.

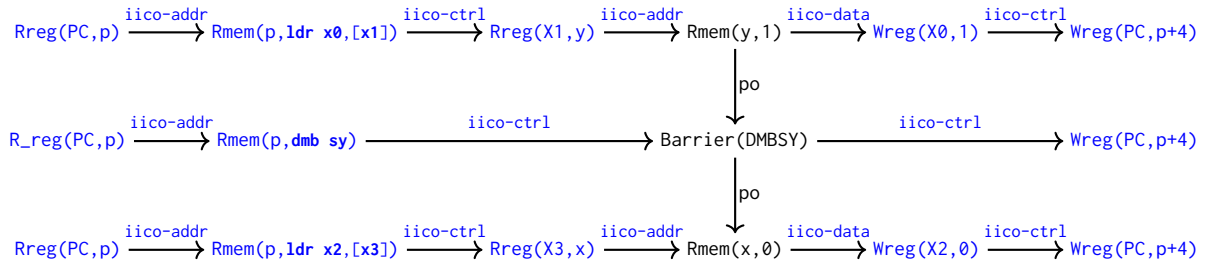


Figure 2.16: Sketch of a completion of reader thread of MP+dmb.sys with implicit events. Nodes and edges in black are original, the ones in blue are the implicit register or instruction fetch events which are part of the ISA definition (with implicit events corresponding to system register accesses and translation elided for brevity).

864 **Well-formedness** Each of the relations of the candidate relations and witness are subject to some well-
 865 formedness constraints. Well-formedness requires that the candidate relations are all properly constructed:
 866 they have the right type, and satisfy some basic relational properties (symmetry, reflexivity, transitivity
 867 and so on) depending on the relation. Figure 2.17 contains the types and some basic well-formedness
 868 properties of the pre-execution relations.

869 Note that a well-formed execution does not necessarily correspond to a consistent execution of the
 870 underlying ISA (see ‘Implicit events and ISA-Consistency’).

Relation	Type	Properties
<code>po</code>	$E \times E$	transitive, asymmetric, irreflexive
<code>loc</code>	$M \times M$	transitive, symmetric, reflexive
<code>ext</code>	$E \times E$	transitive, symmetric, irreflexive
<code>addr, ctrl</code>	$R \times M$	asymmetric, irreflexive
<code>data</code>	$R \times W$	asymmetric, irreflexive
<code>rmw</code>	$R \times W$	asymmetric, irreflexive

Figure 2.17: Non-ISA-dependent well-formedness properties of pre-execution relations.

871 For the existentially-quantified coherence-order and reads-from relations, they are arbitrary, but subject
 872 to the constraints given in Figure 2.18.

$\forall W_1, R_2. \text{rf}(W_1, R_2) \implies \text{loc}(W_1, R_2)$	read and write must be same location
$\forall W_1, R_2. \text{rf}(W_1, R_2) \implies \text{R-VALUE}(R_2) = \text{W-VALUE}(W_1)$	value read matches value written
$\forall W_1, W_2, R_3. \text{rf}(W_1, R_3) \wedge \text{rf}(W_2, R_3) \implies W_1 = W_2$	each read reads-from at most one write
$\forall R_2. \exists W_1. \text{rf}(W_1, R_2)$	every read reads from somewhere
$\forall W_1, W_2. W_1 \neq W_2 \wedge \text{loc}(W_1, W_2)$	
$\implies \text{co}(W_1, W_2) \vee \text{co}(W_2, W_1)$	co is per-location total
$\forall W_1, W_2, W_3. \text{co}(W_1, W_2) \wedge \text{co}(W_2, W_3) \implies \text{co}(W_1, W_3)$	co is transitive
$\forall W_1, W_2. \text{co}(W_1, W_2) \implies \neg \text{co}(W_2, W_1)$	co is antisymmetric
$\nexists W_1. \text{co}(W_1, W_1)$	co is irreflexive

Figure 2.18: Well-formedness conditions of `co` and `rf`.

`R-VALUE` and `W-VALUE` extract the Val from a read or write respectively.

(Hand transcribed from the versions used in `isla-axiomatic`, see §2.5)

873 We say a candidate execution E is *well-formed*, for which we will write $\text{WELL-FORMED}(E)$, if all constraints
874 from Figures 2.17 and 2.18 are satisfied.

Consistency Given an arbitrary pre-execution, that is, a graph with any choice of events and relations, one can define whether or not such a graph corresponds to a valid execution. This can be done by checking that: there exists some witness (`co` and `rf`) such that that candidate is well-formed; that the candidate is consistent with the ISA; and, that does not violate any of the axioms of the model.

$\text{AXIOM-CONSISTENT}(E : \text{Execution}) = \text{see } \S 2.4.2$

$\text{CONSISTENT}(E : \text{Execution}) = \text{WELL-FORMED}(E)$

$\wedge \text{ISA-CONSISTENT}(E)$

$\wedge \text{AXIOM-CONSISTENT}(E)$

$\text{CONSISTENT}(E : \text{Pre-Execution}) = \exists \text{co, rf. CONSISTENT}((E, \langle \text{co, rf} \rangle))$

875 **Program semantics** Architecturally there is no such thing as a ‘program’. Instead, there are only whole
876 machine states. The model then allows us to define what set of configurations are reachable from an initial
877 one, i.e. a ‘program’. There are primarily two ways of representing the initial state in these models: either
878 (1) by only considering executions which are `co`-prefixed by the set of writes corresponding to the initial
879 memory configuration; or, (2) by including some special initial event which other events can read from.
880 The choice of representation does not matter, so we arbitrarily pick the first.

881 Each execution then has a ‘final’ state: the concrete register values for each thread at the end of execution,
882 and the coherence-final write for each location.

We can then define the outcomes permitted by the model by the set of states reachable from the initial state of the program: an outcome is permitted iff there exists a consistent execution, prefixed with the initial writes from the program, whose final state matches that outcome:

$\text{State} \equiv \text{Memory} \times (\text{ThreadId} \rightarrow \text{Registers})$

$\text{FINAL}(E : \text{Execution}) = \text{`Final register and memory state of } E'$

$\text{PREFIXED}(\text{Init} : \text{State}, E : \text{Execution}) = \text{`}E \text{ has co-initial writes corresponding to the initial state'}$

$\text{REACHABLE}(\text{Init} : \text{State}, S : \text{State}) = \exists E : \text{Pre-Execution, co, rf.}$

let $C = (E, \langle \text{co, rf} \rangle)$ in

$\text{PREFIXED}(\text{Init}, C)$

$\wedge \text{CONSISTENT}(C)$

$\wedge S = \text{FINAL}(C)$

883 Giving semantics to an Arm-A program can be done by collecting the set of reachable consistent executions,
884 from an initial machine configuration (program):

$\llbracket P : \text{State} \rrbracket = \{S : \text{State} \mid \text{REACHABLE}(P, S)\}$

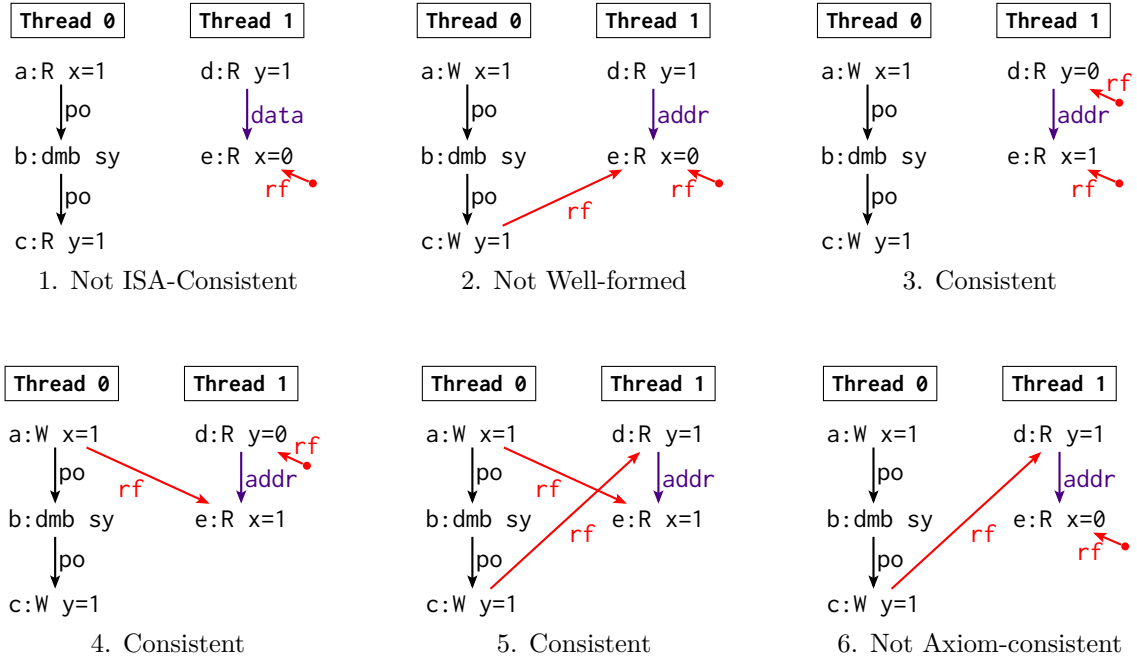


Figure 2.20: Six potential candidate executions for MP+dmb.sy+addr.

885 (Note that this means $\llbracket _ \rrbracket$ is not defined compositionally as a traditional denotational semantics would
886 be, instead, here we have a whole-program consistency check)

887 **An example** Consider the classic MP+dmb.sy+addr litmus test, whose code listing can be found in
888 Figure 2.19. The test has two threads, with two store instructions separated by a barrier in the first, and
889 two loads with a syntactic address dependency between them in the second. Thus, it is an instance of the
890 message-passing shape seen earlier. Figure 2.20 contains six potential candidate executions for this test:

- 891 ▷ Candidate 1 is not consistent with the intra-instruction semantics: it has read events in Thread 0,
892 but the intra-instruction semantics dictate that stores generate write events not read events.
- 893 ▷ Candidate 2 has events consistent with the intra-instruction semantics, but the relations are not
894 consistent with the well-formedness conditions (specifically, rf does not satisfy the ‘read and write
895 must be same location’ constraint), and so this candidate is not well-formed.
- 896 ▷ Candidates 3, 4 and 5, are well-formed, and consistent with the ISA, and consistent with the axioms
897 of the model (given in §2.4.2).
- 898 ▷ Candidate 6 is well-formed, and consistent with the ISA, but not consistent with the axioms.

899 The four well-formed candidate executions listed in Fig-
900 ure 2.20 are the *only* well-formed and ISA-Consistent can-
901 didates for this test. Executions with other events would
902 not be ISA-Consistent; those with co and rf other than
903 those shown would not be well-formed; those with read
904 or write values other than those shown would also not be
905 ISA-Consistent, as those values must have arisen from an
906 execution of the intra-instruction semantics. Only Candi-
907 date 6 has a final state which satisfies the $1:X0=1, 1:X2=0$
908 constraint of the test. Since no candidate satisfying the
909 final state constraint is consistent with the axioms, the
910 test is *forbidden*.

MP+dmb.sy+addr		AArch64
Initial state: 0:X1=x, 0:X3=y, 1:X1=y, 1:X3=x, *x=0, *y=0		
Thread 0	Thread 1	
MOV X0,#1 STR X0,[X1] DMB SY MOV X2,#1 STR X2,[X3]	LDR X0,[X1] EOR X4,X0,X0 LDR X2,[X3,X4]	
Forbidden: 1:X0=1, 1:X2=0		

Figure 2.19: MP+dmb.sy+addr test code listing.

2.4.2 Arm-A axioms

Axiomatic models define *axioms* over candidates, primarily as acyclicity requirements over derived relations over their events. The axioms of the model define which executions are *Axiom-consistent*. Final states from consistent executions are those states that are permitted by the model to be observed on hardware.

Historically, axiomatic models were given as a set of constraints over the derived relations of the model [52, 8]. Contemporary models are described as point-free definitions of relations and acyclicity conditions over them [32, 42]. The derived relations are constructed composing the candidate relations \mathcal{C}_R , and the restricted identity relation (\mathbf{id}_E , for identity over events with label E), using standard relation operators: union (\cup), intersection ($\&$), relation composition (by sequential composition, with $;$), transitive closure ($^+$), and relation inverse ($^{-1}$). The model is then a set of axioms (acyclicity or emptiness) over relations defined in this algebra.

We write these models in the Herd model definition language (often commonly referred to as simply *Cat*), introduced by Alglave et al. [32]. *Cat* is a general language that allows one to express first-order quantifier-free relations, in a relatively concise syntax, using a set of built-in relations and relational operators. Values in *Cat* are either sets of events, or relations (sets of pairs of events). *Cat* lets the user define either sets of events, or relations over events, using the usual set of set and relational operators, with some custom syntax, reproduced here for quick reference:

- ▷ R^+ for the transitive closure (one-or-more repetitions) of R .
- ▷ $[E]$ for the identity over events with label E , corresponding to the mathematical relation \mathbf{id}_E ,
- ▷ $[E1|E2|\dots]$ for the set of events with labels which match $E1$ or $E2$ or so on.
- ▷ $\text{domain}(R)$ and $\text{range}(R)$ give the sets of events that are the domain and codomain of a relation R .
- ▷ $(E1 * E2)$, is the relation formed by the cartesian product of sets of events with labels $E1$ and $E2$, that is, the mathematical relation $\text{range}(\mathbf{id}_{E1}) \times \text{range}(\mathbf{id}_{E2})$. $E1$ or $E2$ can be substituted with an underscore which acts as a wildcard that matches events with any label.
- ▷ id for the generalised identity relation over events, which corresponds to $\mathbf{id}_;$
- ▷ $R?$ as a shorthand for relation option, equivalent to $R \mid \text{id}$.

The original *herdtools* *Cat* language and the *isla-axiomatic* *Cat*-like model language have diverged over time, but the features described in this section remains common to both.

An Arm-A *Cat* model A reformulation of the original non-mixed-size multi-copy-atomic Armv8-A model from 2018 [7, 53], can be found in Figure 2.21. The other models presented in this thesis will be extensions to the one presented here. Note that this particular presentation of the model is slightly different from the original, with the transitive relations over barriers split into multiple edges explicitly relating events to barriers, and lifting `coi` and `fri` into `obs`. Although equivalent to the original, this presentation will be easier to extend, the reason for which will become apparent later on. Additionally, the current official Arm models have diverged from the original model this one is based on, either through the addition of new features (mixed-size, memory tagging extensions, and so on), or through iterative refactors of the model over time [42]. An *isla-axiomatic-executable* version of the model can be found at https://github.com/remns-project/system-semantics-arm-axiomatic-models/blob/main/models/aarch64_interface.cat.


```

1  (* observed by *)
2  let obs = rfe | fr | co
3
4  (* dependency-ordered-before *)
5  let dob =
6      addr | data
7      | ctrl; [W]
8      | addr; po; [W]
9      | (ctrl | (addr; po)); [ISB]
10     | (addr | data); rfi
11
12 (* atomic-ordered-before *)
13 let aob = rmw
14     | [range(rmw)]; rfi; [A | Q]
15
16 (* barrier-ordered-before *)
17 let bob = [R] ; po ; [dmbld]
18     | [W] ; po ; [dmbst]
19     | [dmbst]; po; [W]
20     | [dmbld]; po; [R|W]
21     | [ISB]; po; [R]
22     | [L]; po; [A]
23     | [A | Q]; po; [R | W]
24     | [R | W]; po; [L]
25 (* Ordered-before *)
26 let ob1 = obs | dob | aob | bob
27 let ob = ob1+
28
29 (* Internal visibility
   requirement *)
30 acyclic po-loc | fr | co | rf
   as internal
31
32 (* External visibility
   requirement *)
33 irreflexive ob as external
34
35 (* Atomic: Basic LDXR/STXR
   constraint to forbid
   intervening writes. *)
36 empty rmw & (fre; coe) as atomic

```

Figure 2.21: Armv8-A multi-copy atomic ‘user’ axiomatic model.

950 The Cat model relies on a set of built-in event sets and relations, these are:

Events	Relations
R Reads	po,rmw program-order and read-modify-write
W Writes	po-loc po between same-location events
M Explicit memory events (R W)	addr/ctrldata dependencies
951 A Read-acquire	co/rf Witness
L Write-release	rfi/coi internal (within thread) rf/co
Q Weak read-acquire	rfe/coe external (across threads) rf/co
F Fences (barriers)	id identity
ISB Instruction synchronization barrier	
dmbXY Memory barrier with kind XY	

952 **The axioms** The Arm-A model is made up of three axioms: **external** (line 33), which asserts acyclicity
953 of a global ordered-before relation, capturing most of the ordering constraints of the Arm memory model;
954 the **internal** axiom (line 30), sometimes called ‘SC-per-location’, which ensures that when restricted to a
955 single location the accesses are consistent with an SC semantics; and, the **atomic** axiom (line 36) which
956 asserts that there can be no same-location writes interposing between events of an atomic action.

957 **Ordered-before** The main relation, ordered-before (ob), is defined on line 27 as the transitive closure of
958 the union of a set of auxiliary ordering relations. These auxiliary relations are:

- 959 ▷ observed-by (obs, line 2), which orders events after the events they observe the effect of, namely,
960 writes must happen before other-thread reads which read from them.
- 961 ▷ dependency-ordered-before (dob, line 5), which orders events which must not be re-ordered due to
962 syntactic dependencies in the original source program.
- 963 ▷ atomic-ordered-before (aob, line 13) which asserts that the read of an atomic read-modify-write
964 happens before the write, and that acquire reads of an atomic write are ordered.

965 ▷ barrier-ordered-before (bob, line 17) between events where there is an intervening barrier instruction
 966 ordering them.

967 A candidate execution with a cycle in ordered-before is forbidden. For example, in the following
 968 MP+dmb.st+addr test, whose code listing and event diagram for the forbidden execution can be found in
 969 Figure 2.22.

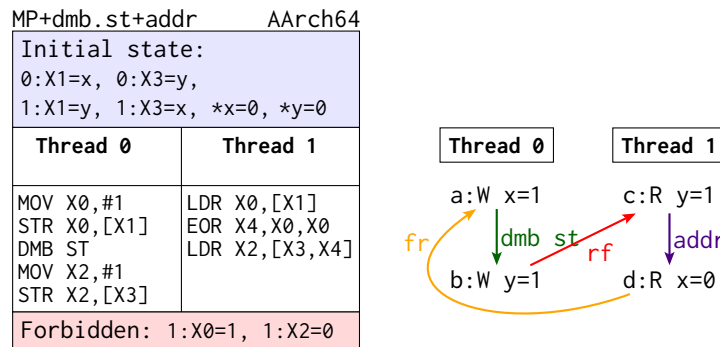


Figure 2.22: MP+dmb.st+addr test code listing and execution diagram.

970 The interesting candidate execution is the one that has the final state $1:X0=1 \wedge 1:X2=0$. It is this
 971 candidate that is shown in Figure 2.22 (right). Being a message-passing (MP) shape, it is characterised
 972 by a po-rfe-po-fre¹ cycle. In particular, it has the following cycle:

- 973 ▷ a dmbst b
- 974 ▷ b rfe c
- 975 ▷ c addr d
- 976 ▷ d fr a

977 This cycle is forbidden in the Arm model, as each of the relations are contained in ob, and a cycle in ob is
 978 forbidden by the external axiom:

- 979 ▷ ([W]; dmbst; [W]) is in bob, which is in ob.
- 980 ▷ rfe is in obs, which is in ob.
- 981 ▷ addr is in dob, which is in ob.
- 982 ▷ fr is in obs, which is in ob.

983 **Internal and atomic axioms** The other axioms of the model forbid behaviours that the ordered-before
 984 acyclicity check does not recognise, such as non-SC behaviours for single locations, or supposedly
 985 atomic actions (such as exclusives or read-modify-writes) which were interrupted by an intervening
 986 write. Figure 2.23 contains two example tests, a coherence test forbidden by the internal axiom and an
 987 LB-shaped atomic increment failure forbidden by the atomic axiom.

988 Note that this is not the only possible presentation of the model. A separate internal/SC-per-location
 989 axiom is classic, but the current official herdttools version of the Arm model has separate axioms for
 990 each of the forbidden coherence shapes [54]. The external axiom usually considers a partially-ordered
 991 ordered-before relation built from smaller primitive relations, as was presented here, but other formulations
 992 sometimes pick some linearisation of some total order, equivalent to but more operational in presentation
 993 than the one presented here.

2.5 The isla-axiomatic tool

995 Throughout this work we will use the isla-axiomatic tool [36] to implement executable versions of our
 996 axiomatic models. The isla-axiomatic tool uses the full ASL specification of the Arm ISA, converted to
 997 Sail. The generation of candidates then uses whole machine states, including all instruction fetch and
 998 translation table walks as real memory accesses.

¹More precisely a PodWW Rfe PodRR Fre cycle in diyone syntax [34].

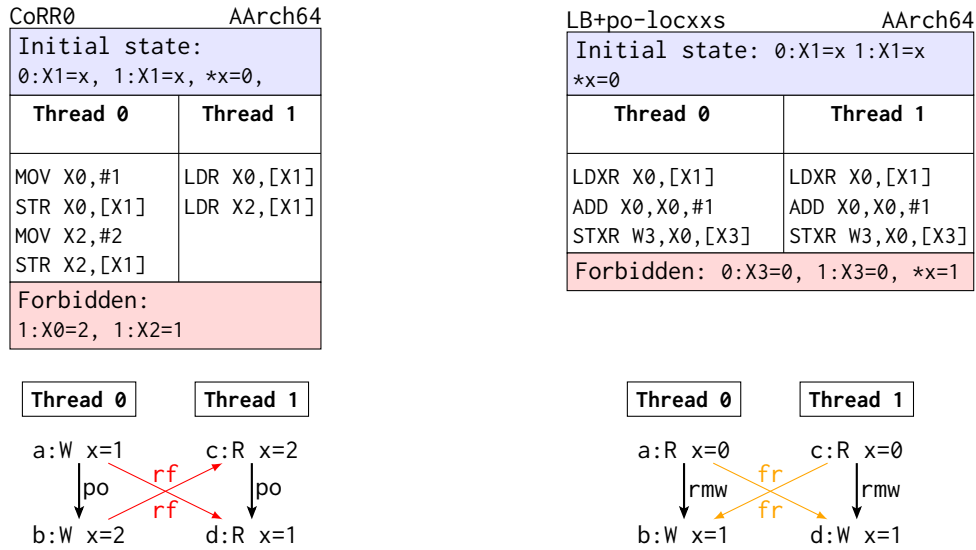


Figure 2.23: Two tests forbidden by the other axioms. On the left, a variation on coherence which relies on po-loc and so is forbidden by the internal axiom. On the right, an atomic increment that failed to atomically update the location, forbidden by the atomic axiom.

```

1  function Step() {
2    if pending interrupts then {
3      TakePendingInterrupt();
4    };
5
6    let pc = Read_reg(PC);
7
8    let opcode = \
9      Read_mem(
10     ReadKind_IFETCH,
11     pc, 4);
12     // magic opcode not part of ISA
13     if opcode == 0xfe1dead {
14       EndOfTrace();
15     };
16
17     let instr = ArmASL_Decode(opcode);
18
19     ArmASL_Execute(instr);
20
21     Write_reg(PC, pc+4)
22   }

```

Figure 2.24: Extract of the Arm top-level step function.

999 Using `isla-axiomatic` allows us to use the Arm ASL definitions which already exist (for instruction
1000 fetching, decoding, and translation table walks in particular), giving us those fundamental executions ‘for
1001 free’ for those features, and enabling us to focus on modelling the concurrent aspects of them.

1002 **isla-axiomatic candidates** Underpinning the `isla-axiomatic` tool is `isla`, a generic symbolic evaluator
1003 for Sail programs [36]. `isla-axiomatic` uses `isla` to generate candidate executions, by producing traces
1004 of Sail outcomes for each thread, with concrete control flow but potentially symbolic values for reads
1005 and writes. `isla-axiomatic` then produces the relevant dependency relations (which it does in an ad-hoc
1006 way), then applies a restriction to the events of the traces (discarding all events except reads, writes and
1007 barriers for the base model), and takes the cartesian product of these restricted traces of events for each
1008 thread; the result is precisely the set of well-formed pre-executions (but with symbolic values).

1009 There is a fetch-decode-execute loop for each thread, which sequentially fetches the next instruction and
1010 runs the Sail (converted from ASL) decode and execute functions, until a pre-determined point is reached
1011 (usually a particular ‘end-of-test’ opcode) which signifies the end of the trace of that instruction. A sketch
1012 of the top-level fetch-decode-execute function from our Arm Sail model¹ is given in Figure 2.24.

1013 During symbolic evaluation of the Sail code, when a branch’s condition is symbolic and not constrained to
1014 one of true or false, the symbolic execution forks. This gives a tree of traces of outcomes for each thread,
1015 with concrete opcodes and register names, but with constrained symbolic values.

¹<https://github.com/rems-project/sail-arm/blob/master/arm-v9.3-a/src/step.sail#L217>

1016 We can then use this as an executable oracle for litmus tests. By taking the well-formed pre-executions
1017 generated from those symbolic traces, `isla-axiomatic` can produce a single SMT problem for each
1018 candidate whose satisfiability encodes whether the candidate is consistent. It does this by creating SMT
1019 definitions of: the events from the pre-execution with constraints on symbolic values; the candidate
1020 relations (in particular, coherence-order and reads-from); the axioms of the model and any auxiliary
1021 relations from the Cat model; with the final assertion from the litmus test. Giving this SMT problem to
1022 an off-the-shelf SMT solver (such as Z3) allows automatic consistency checking; if the SMT solver can
1023 find a satisfying assignment of the symbolic values, then the execution is allowed; if the SMT solver says
1024 it is unsatisfiable then the execution is either forbidden by the axioms, or does not satisfy the constraint
1025 on the final state. If all executions when compiled to SMT are unsatisfiable then the test as forbidden.

1026 2.5.1 ISA/concurrency interface

1027 *This section is based on in-progress work with Thibaut Pérami, Alasdair Armstrong, Thomas Bauereiss,*
1028 *and Peter Sewell.*

1029 As `isla-axiomatic` uses the full ISA outcomes, the model should be able to utilise any information
1030 exposed in the Sail outcome type. To achieve this the `isla-cat` language is extended with the structs and
1031 enums from the Sail definition, and an *accessor* construct allowing the model writer to define event sets
1032 predicated on the values of fields of the underlying Sail structs.

1033 As previously mentioned, each event in an `isla-axiomatic` candidate execution corresponds to an outcome
1034 in the trace of the intra-instruction semantics. The outcomes then form the interface between the sequential
1035 ISA semantics and the concurrency model. The current Sail ISA/concurrency interface is defined in
1036 https://github.com/rems-project/sail/tree/sail2/lib/concurrency_interface.

1037 For example, the Arm Sail model contains the `sail_barrier` outcome¹ :

```
1038     outcome sail_barrier : 'barrier -> unit
```

1039 Each architecture's Sail specification can then instantiate the 'barrier type variable with architecture-
1040 specific data. For instance, in Armv9-A the 'barrier type is instantiated with a custom Barrier type²,
1041 given in Figure 2.25, which contains a translation of the Arm barrier kind datatype in the ASL/Sail
1042 specification.

1043 Then the Sail Arm specification can use the `sail_barrier` outcome to generate events in the trace. For
1044 example, the `DataSynchronizationBarrier` function is the function which the specification calls on execution
1045 of a DSB instruction. It is an uninterpreted function in the specification. The Sail model implements that
1046 function to output a `sail_barrier` effect³, which generates a barrier event in the trace when executed:

```
1047 1 function DataSynchronizationBarrier (domain, types, nXS) = {  
1048 2     sail_barrier(  
1049 3         Barrier_DSB(  
1050 4             struct { domain = domain, types = types, nXS = nXS }  
1051 5         )  
1052 6     )  
1053 7 }
```

¹https://github.com/rems-project/sail/blob/0.18/lib/concurrency_interface/barrier.sail#L75

²<https://github.com/rems-project/sail-arm/blob/interface-v9/arm-v9.3-a/src/interface.sail#L286>

³<https://github.com/rems-project/sail-arm/blob/interface-v9/arm-v9.3-a/src/stubs.sail#L105>

```

1  enum MBReqDomain = {
2      MBReqDomain_Nonshareable,
3      MBReqDomain_InnerShareable,
4      MBReqDomain_OuterShareable,
5      MBReqDomain_FullSystem
6  }
7
8  enum MBReqTypes = {MBReqTypes_Reads, MBReqTypes_Writes, MBReqTypes_All}
9
10 struct DxB = {
11     domain : MBReqDomain,
12     types : MBReqTypes,
13     nXS : bool
14 }
15
16 union Barrier = {
17     Barrier_DSB : DxB,
18     Barrier_DMB : DxB, // The nXS field is ignored from DMBs
19     Barrier_ISB : unit,
20     Barrier_SSBB : unit,
21     Barrier_PSSBB : unit,
22     Barrier_SB : unit,
23 }
24
25 instantiation sail_barrier with
26     'barrier = Barrier

```

Figure 2.25: Arm interface: instantiation of barriers.

2.5.2 Extended Cat with Sail interface

The extended `isla-cat` language is very similar to the original `Cat` language but with some differences. Since `isla-axiomatic` does not support mutually recursive bindings, procedures, or inline function definitions, we will not use them in our models.

Unlike `Cat`, `isla-axiomatic` does not define a large collection of built-in relations and sets. Instead, it adds *accessors*: point-free declarations which define functions over events. Accessors can access the fields of the underlying `Sail` structures to allow the model author to define their own relations and sets based on the underlying ISA definitions.

For example, the `Armv9-A` accessor for barrier access types matches on the `Barrier` union we saw earlier, and if it is one of `Barrier_DMB` or `Barrier_DSB` it extracts the `.types` field from its `DxB` struct, and otherwise returns the default value for that type. The `isla-cat` definition of such an accessor is given below:

```

1  accessor barrier_types: MBReqTypes = .match {
2      Barrier_DMB => .types,
3      Barrier_DSB => .types,
4      _ => default
5  }

```

These accessors can be used in simple function declarations, using the `isla-cat define` command. For example, the `Armv9-A` model defines the `F` (*fence*) event type and the various Arm barrier event kinds (`dmb ld,dmb sy, ...`) with accessors. An extract of the `isla-cat` definition for `Armv9-A`¹, for the definition of an example barrier event (`dmbld`, the event set that includes all barrier events that are at least as strong as a `DMB.LD` instruction), is given in Figure 2.26.

¹Full definition can be found at <https://github.com/rem-s-project/system-semantics-arm-axiomatic-models/blob/main/models/armv9-interface/barriers.cat>

```

1  accessor F: bool = is sail_barrier
2
3  define has_barrier_type(ev: Event, t: MBReqTypes): bool =
4      (barrier_types(ev) == t)
5
6  accessor is_DxB: bool =
7      .match {
8          Barrier_DMB => true,
9          Barrier_DSB => true,
10         _ => false
11     }
12
13  accessor is_DMB: bool =
14      .match {
15          Barrier_DMB => true,
16          _ => false
17     }
18
19  define ArmBarrierRM(ev: Event): bool =
20      is_DxB(ev) & has_barrier_type(ev, MBReqTypes_Reads)
21
22  define DMB(ev: Event): bool =
23      F(ev) & is_DMB(ev)
24
25  define DMBLD(ev: Event): bool = DMB(ev) & ArmBarrierRM(ev)
26
27  define dmbld(ev: Event): bool =
28      (* see full code for definitions of dmbsy and dsbld *)
29      DMBLD(ev) | dmbsy(ev) | dsbld(ev)

```

Figure 2.26: Arm interface: definition of barrier accessors for an example barrier kind.

Instruction fetch

1075
1076
1077 *This part is based on: ARMv8-A system semantics: instruction fetch in relaxed architectures [35] by Ben*
1078 *Simner, Shaked Flur, Christopher Pulte, Alasdair Armstrong, Jean Pichon-Pharabod, Luc Maranget, and*
1079 *Peter Sewell. Published in the proceedings of the 29th European Symposium on Programming (ESOP,*
1080 *2020).*

Relaxed instruction fetching

We now describe the main instruction fetch phenomena and architecture design questions for Arm-A. As usual, this will be done through handwritten litmus tests, which we will use to guide model design later on.

Chapter contents

3.1	Introduction	41
3.2	Industry practice and the existing Arm prose	42
3.3	Modifiable instructions	44
3.4	Coherence	46
3.4.1	Instruction-to-Instruction coherence	46
3.4.2	Data-to-Instruction coherence	47
3.4.3	Instruction-to-Data coherence	48
3.5	Cross-thread synchronisation	49
3.6	Cache maintenance	50
3.6.1	Synchronisation on a single thread	50
3.6.2	Broadcast cache maintenance	51
3.7	Dependencies	53
3.7.1	Address dependencies	53
3.7.2	Control dependencies	53
3.8	Multi-Copy Atomicity	54
3.9	More on instruction caches	54
3.10	Points of Unification and Coherency	55
3.10.1	Late unification	56
3.10.2	Promotion	57
3.11	Cleans and invalidates are like reads and writes	57
3.11.1	Cleans are similar to reads	57
3.11.2	IC invalidates are not quite like writes	57
3.11.3	DC and IC address speculation	58
3.11.4	DC might be to same address	59
3.11.5	DC ordering with respect to other memory accesses	60
3.12	Same-cache-line ordering	61
3.13	Mixed-size instruction fetching	62
3.14	Cache type strengthening: IDC and DIC	63
3.14.1	IDC	63
3.14.2	DIC	63
3.15	Related Work	64

1121 3.1 Introduction

1122 Self-modifying code is a software pattern relied on by nearly all software, but only explicitly managed by
1123 few: mostly systems software, such as dynamic loaders, operating system kernels, and hypervisors; and
1124 also some usermode, like just-in-time (JIT) compilers. This software forms part of the security-critical
1125 computing base, currently trusted but not trustworthy, that is especially in need of verification, and which
1126 will require a precise and well-validated definition of the architectural abstraction.

1127 The semantics required for self-modifying code, of instruction fetch and cache maintenance, are areas where
1128 microarchitectural optimisations can have surprising programmer-visible effects, especially in the concurrent
1129 context. Previous work has scarcely touched on this: none of seL4 [55], CertiKOS [56, 57], Komodo [58],
1130 nor the works of Guanciale et al. [59], nor Baudmann et al. [60], address realistic architecture concurrency,
1131 and they use (at best) idealised models of the sequential systems architecture. The CakeML [61, 62] and
1132 CompCert [63] verified compilers target only sequential user-mode ISA fragments, without self-modifying
1133 code. Previous attempts at verification of self-modifying code have typically focused on MIPS or x86,
1134 such as in the works of Cai et al. and Myreen [64, 65]. However, those architectures have a very different
1135 programmer model than Arm presents, not requiring explicit instruction cache maintenance.

1136 In this part we focus on instruction fetch, and its required cache maintenance, for Arm-A. The ability to
1137 execute code that has previously been written to data memory is fundamental to computing: fine-grained
1138 self-modifying code is now rare, and rightly deprecated, but program loading, dynamic linking, JIT
1139 compilation, debugging, and OS configuration, all rely on executing code from data writes. However,
1140 because these are relatively infrequent operations, hardware designers have been able to optimise by
1141 partially separating the instruction and data paths, with distinct instruction caching, which by default may
1142 not be coherent with data accesses. This can introduce programmer-visible behaviour analogous to that
1143 of user-mode relaxed-memory concurrency, and requires specific additional synchronisation to correctly
1144 pick up code modifications. Exactly what these are was not entirely clear in the Arm-A architecture text
1145 at the time this work was done (up to version D.a [66]).

1146 We clarify this situation, developing precise abstractions that bring the instruction-fetch part of Arm-A
1147 system behaviour into the domain of rigorous semantics. We aim thereby to enable future work on
1148 system software verification using the techniques of programming languages research: program analysis,
1149 model-checking, program logics, and so on. At the time of this work Arm intended to officially incorporate
1150 a version of the model into their architecture, [67]. Since then, Arm have developed an instruction fetch
1151 model as part of the official Arm memory model [68, B2.3]. Detailed comparison of that model with the
1152 one presented here will be an important topic for future work, although we do not believe the architectural
1153 intent has changed.

1154 **Overview** In this chapter, we begin by recalling the informal architectural guarantees that the Arm-A
1155 architecture provides, and the ways in which real-world software systems such as Linux, the JavaScript
1156 and WebAssembly JITs, and other language implementations modify instruction memory. We then survey
1157 the fundamental phenomena and architecture design questions with a series of examples, and explore the
1158 interactions between instruction fetching, cache maintenance and the ‘usual’ relaxed memory stores and
1159 loads, showing that instruction fetches are more relaxed, and how even fundamental coherence guarantees
1160 for data memory do not apply to instruction fetches.

1161 We give an operational semantics for Arm instruction fetch and cache maintenance (Ch.4) in an abstract-
1162 microarchitectural style (following the Flat model of Flur, Pulte et al. [7], cf. §2.3) capturing the
1163 architectural intent as we understand it. We make the operational model executable as a test oracle by
1164 integrating it into the RMEM tool [51], with optimisations that make it possible to exhaustively execute
1165 example litmus tests.

1166 We give a more concise presentation of the model (Ch.5) in an axiomatic style (in the style of Alglave et
1167 al.’s herd models [32], as an extension to the official Armv8 model of Deacon [7], cf. §2.4), and intended to
1168 be extensionally equivalent to the aforementioned operational semantics. We give an executable-as-a-test-
1169 oracle formulation of the model, in an extension to the isla-axiomatic tool, which with Armstrong and
1170 Campbell we extend to support instruction fetching litmus tests.

1171 We validate all this (Ch.6), in two ways: by extensive discussion with Arm staff and systems software
1172 engineers, and by experimental testing of hardware behaviour on a selection of Armv8-A cores designed
1173 by multiple vendors. We run tests on hardware with a mild extension of the Litmus tool [33, 69]. We

1174 then compare hardware and the two models on a suite of 1456 tests, automatically generated with an
1175 extension of the diy tool [70]. We also check the operational and axiomatic models for regressions against
1176 the sets of previous non-ifetch tests. We found no regressions and no test which distinguishes the models,
1177 and all data was consistent with hardware observations, except for one case where our testing uncovered a
1178 hardware bug on a Qualcomm device.

1179 **Caveats and Limitations** Our operational semantics are integrated with a substantial fragment of the Sail
1180 Armv8-A ISA (similar to that used for CakeML [71]), but not yet with the full ISA model [45, 10, 11, 72];
1181 this is a matter of additional engineering and is future work. We do not handle the interaction between
1182 instruction fetch and mixed-size accesses, or other variants of the cache maintenance instructions, e.g. those
1183 used for interaction with DMA engines or variants by set or way instead of by virtual address. Finally,
1184 while the equivalence between our operational and axiomatic models is validated experimentally, we do
1185 not have a formal proof of equivalence. A proof of this equivalence will be essential in the long term,
1186 but represents a major step and substantial work itself: the complexity makes mechanisation essential,
1187 but the operational model (in all its scale and complexity) has not yet been subject to mechanised proof.
1188 Without instruction fetch, a non-mechanised proof was the main result of an entire PhD thesis [6], and we
1189 expect the addition of instruction fetch to require global changes to the argument.

1190 3.2 Industry practice and the existing Arm prose

1191 Computer architecture relies on a host of sophisticated techniques for performance, including buffering,
1192 caching, prediction and prefetching, and pipelining. For the normal memory reads and writes of ‘user-mode’
1193 concurrency, the programmer-visible relaxed-memory effects largely arise from store buffering and from
1194 out-of-order and speculative pipeline behaviour, not from the cache hierarchy (though some IBM POWER
1195 phenomena do arise from the interconnect, and from late processing of cache invalidates).

1196 At first sight, one might expect instruction fetches to act like other memory reads. However, writes to
1197 instruction memory are relatively rare, so hardware designers have adopted much more aggressive caching
1198 mechanisms specifically for those accesses. The Arm architecture carefully does not mandate exactly what
1199 these may be, permitting a wide range of possible hardware implementations. For example, a typical
1200 high-performance Arm processor might have per-core separate L1 instruction and data caches, above a
1201 unified per-core L2 cache and an L3 cache shared between cores. There may also be additional structures,
1202 e.g. per-core fetch queues, loop buffers, and caching of decoded micro-ops. Figure 3.1 shows a typical
1203 micro-architectural design: that of the Arm Cortex-A53, with independent per-thread instruction and
1204 data caches, which unify into a global cache before memory. Data flows out of the core into the L1 data
1205 cache, and then from the data cache to the instruction cache or out to memory.

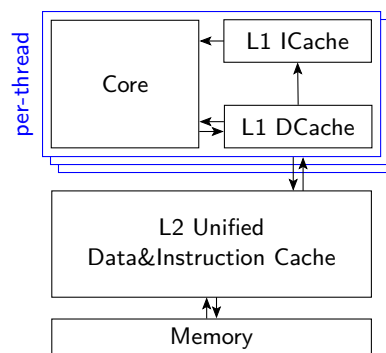


Figure 3.1: Block diagram of the Arm Cortex-A53 [73], with simplified data and instruction flow [74].

1206 **Modifying code and maintaining coherence** In usermode models the caches are, aside from performance
1207 implications, invisible to the programmer, with the hardware cache protocol managing them automatically.
1208 In contrast, the caching of instruction data exposes details to the programmer that would otherwise be
1209 invisible: such as the cache line size (as caches may cache arbitrarily large ‘lines’ of memory at a time)
1210 and physical hierarchy of the caches.

1211 This is because instruction caching is not necessarily coherent with data memory accesses¹ [66, B2.4.4
1212 (B2-114)]:

the architecture does not require the hardware to ensure coherency between
instruction caches and memory

1213

1214 Hence, programmers must use the explicit cache maintenance instructions [66]:

If software requires coherency between instruction execution and memory, it
must manage this coherency using Context synchronization events and cache
maintenance instructions.

1215

1216 The manual further gives a sufficient sequence [66]:

```
; Coherency example for data and instruction accesses [...]
; Enter this code with <Wt> containing a new 32-bit instruction,
; to be held in Cacheable space at a location pointed to by Xn.
STR Wt, [Xn]; Store new instruction
DC CVAU, Xn ; Clean data cache by virtual address (VA) to PoU
DSB ISH     ; Ensure visibility of the data cleaned from cache
IC IVAU, Xn ; Invalidate instruction cache by VA to PoU
DSB ISH     ; Ensure completion of the invalidations
ISB        ; Synchronize the fetched instruction stream
```

1217

1218 At first sight, this may be entirely mysterious. This and the following chapters establish precise semantics
1219 for each of the above instructions, explaining why each is required. However, a rough intuition for each is:

- 1220 1. The DC CVAU, Xn cleans this core's data cache for address Xn, pushing the new write far enough down
1221 the hierarchy for any instruction fetch that misses in the instruction cache to be guaranteed to see
1222 the new value. This point is the *Point of Unification* (PoU) and is usually the point where the
1223 instruction and data caches become unified (L2 for most modern devices).
- 1224 2. The DSB ISH waits for the clean to have happened before letting the later instructions execute
1225 (without this, the sequence itself can execute out-of-order, and the clean might not have pushed the
1226 write down far enough before the instruction cache is updated). The ISH makes this specific to the
1227 *Inner Shareable Domain*: the processor itself, not the system-on-chip. We do not model shareability
1228 domains in this work, so this is equivalent to a DSB SY.
- 1229 3. The IC IVAU, Xn invalidates any entry for that address in the instruction caches for all cores, forcing
1230 any future fetch to miss in the instruction cache, and instead read the new value from the data
1231 memory hierarchy.
- 1232 4. The second DSB ISH waits for the cache invalidation to complete.
- 1233 5. The final ISB flushes this core's pipeline, forcing a re-fetch of all program-order-later instructions.

1234 Central to this sequence is the concept of *cache maintenance*. Caches will fetch data automatically, but
1235 may require explicit operations to remove old cached values. Cache maintenance operations can generally
1236 be split into one of two kinds:

- 1237 ▷ Cleans force a write-back of a cache line, pushing any potential cached copies further down the
1238 cache hierarchy.
- 1239 ▷ Invalidations remove cached copies of a whole line.

1240 From the programmer's perspective, invalidations are destructive: if the removed cached line's data did
1241 not exist further down the cache hierarchy, or in memory, then the data may be lost entirely. However,
1242 cleans push data further out thereby making it *more* widely visible. For instruction cache maintenance,
1243 only invalidation is provided, but for data cache maintenance the programmer can choose whether to do
1244 a clean, an invalidate, or both; and whether the maintenance takes effect to the Point of Unification or
1245 the Point of Coherency (see §3.10 for an explanation of what the different points mean). Arm therefore
1246 provide a large collection of cache maintenance instructions, of which the most relevant for this part are:

¹Version J.a of the Arm architecture reference manual includes the word 'not' here, which is a typographical error.

Instruction	Operation
DC CVAU	Clean Data&Unified Caches by VA to PoU
DC IVAC	Invalidate Data&Unified Caches by VA to PoC
DC CVAC	Clean Data&Unified Caches by VA to PoC
DC CIVAC	Clean&Invalidate Data&Unified Caches by VA to PoC
IC IVAU	Invalidate Instruction Caches by VA to PoU
IC IVAC	Invalidate Instruction Caches by VA to PoC
IC IALLU	Invalidate Local Instruction Cache to PoU
IC IALLUIS	Invalidate All Instruction Caches to PoU

We discuss more about the relationship between these cache maintenance operations in §3.10.2.

Some hardware implementations provide extra guarantees, rendering the DC or IC instructions unnecessary. Arm allow software to discover this in an architectural way, by reading the CTR_EL0 register's DIC and IDC fields, described in more detail in §3.14.

Concurrent modification and instruction fetch require the same sequence, with an ISB on each thread that executes the new instructions, and the rest of the sequence on the modifying thread [66, B2.2.5 (B2-94)]. Concurrent modification without synchronisation is restricted to particular instructions (B (branch), BL (branch-and-link), BRK (break), SMC, HVC, SVC (secure monitor, hypervisor, and supervisor calls), ISB, and NOP), otherwise there could be *constrained unpredictable behaviour*: ‘any behavior that can be achieved by executing any sequence of instructions that can be executed from the same Exception level’. All this gives some guidance for programmers, but leaves the exact semantics of instruction fetch and those cache maintenance instructions unclear.

Linux has many places where it modifies code at runtime: in boot-time patching of *alternatives*, modifying kernel code to specialise it to the particular hardware being run on; when the kernel loads code (e.g. when the user calls `dlopen`); and in the `ptrace` system call, used e.g. by the GDB debugger to patch arbitrary instructions with breakpoints at runtime. In Google's *Chrome* web browser, its WebAssembly and JavaScript just-in-time (JIT) compilers write new code during execution and modify existing code at runtime. In the JavaScript JIT, this modification happens inside a single thread and so is relatively straightforward. The WebAssembly case is more complex, as one thread is modifying the code being concurrently executed by another.

In practice, software typically does not use the above sequence verbatim. For example, when synchronising a range of addresses all at once, the software may group the DCs and ICs of different addresses together. Additionally, the final ISB may be subsumed by other instruction synchronisation e.g. from exception entry or return. Software threads may also be migrated (by the OS or hypervisor) from one hardware thread to another, potentially interrupting such an instruction cache maintenance sequence. Moreover, for security reasoning, we have to be able to bound the possible behaviour of arbitrary code. For all these reasons, we must consider the effect of each instruction individually and how they compose, and cannot simply assume a canned sequence.

The problem we face is to give such a semantics that correctly defines behaviour in arbitrary concurrent contexts, that captures the Arm architectural intent, that is strong enough for software, and that abstracts from the variety of hardware implementations (e.g. with differing cache structures) that the architecture intends to allow – but which programmers should not have to think about.

3.3 Modifiable instructions

As was mentioned in §3.2, concurrent modification and execution is only permitted if the original and modified instructions are *concurrently modifiable*, which is defined as: simple branches, supervisor/hypervisor/secure monitor calls, the ISB (instruction synchronisation) barrier, the BRK (breakpoint) instruction, and NOP. Otherwise, the architecture permits *constrained unpredictable* behaviour, meaning that the resulting machine state could be anything that would be reachable by arbitrary instructions at the same exception level. Stronger constraints for constrained unpredictable is an area under investigation by Arm.

The following **W+F** test (Figure 3.2, p.45) illustrates this.

W+F		AArch64
Initial state: 0:W0="SUB X0,X0,#1", 0:X1=1		
Thread 0	Thread 1	
STR W0,[X1] //modify Thread 1 at 1	1: ADD X0,X0,#1 //initial code	
Allowed: constrained-unpredictable final state		

Figure 3.2: Code listing for test W+F.

1288 In this test, Thread 0 writes to the code that Thread 1 is executing, overwriting the `ADD X0,X0,#1`
 1289 instruction with the 32-bit encoding of the `SUB X0,X0,#1` instruction. If the fetch were atomic, the
 1290 outcome of this test would be the result of executing either the `ADD` or the `SUB` instruction. However,
 1291 because at least one of those is not a ‘concurrently modifiable’ instruction (not in the set of atomically-
 1292 fetchable instructions given previously), Thread 1 has constrained-unpredictable behaviour and the final
 1293 state is very loosely constrained. Note, however, that this is nonetheless much more constrained than the
 1294 C/C++ whole-program undefined behaviour in the presence of a data race: unlike C/C++, a hardware
 1295 architecture has to define a useful envelope of behaviour for arbitrary code, to provide guarantees for the
 1296 rest of the system when one user thread has a race.

1297 **Debuggers and breakpoints** One challenge in the definition as given by Arm is that it forbids replacing
 1298 arbitrary instructions with breakpoints concurrently. Other architectures (such as IBM Power) simply
 1299 require that at least one of the instructions is concurrently modifiable, not both.

1300 In practice, debuggers replace instructions with breakpoints (the `BRK` instruction) regardless. Further
 1301 work is required to investigate whether a strengthening could be made to the Arm architecture to permit
 1302 this in general.

1303 **Conditional branches** In version D.a (and earlier) of the Arm architecture reference manual, it made
 1304 clear that, for branches with conditions (`B.cond`) which are overwritten by other `B.cond` instructions, the
 1305 Arm architecture provided a specific non-single-copy-atomic fetch guarantee: that the execution will be
 1306 consistent with either the old or new target, with either the old or new condition [66, B2-94]. In version E.a,
 1307 this condition was removed entirely, meaning `B.cond` instructions were not permitted to be concurrently
 1308 updated at all [75, B2-112]. In version G.b, `B.cond` was added to the list of concurrently-modifiable
 1309 instructions, once more permitting replacement of (and with) a `B.cond` instruction [76, B2-130], with the
 1310 stronger semantics that one will see either the old instruction or the new instruction entirely.

W+F+branches		AArch64
Initial state: 0:W0="B.NE h", 0:X1=1		
Thread 0	Thread 1	
STR W0,[X1]	1: B.EQ g	
Final state: execute "B.NE g"		

Figure 3.3: Code listing for test W+F+branches.

1311 For example, the `W+F+branches` test (Figure 3.3) overwrites a `B.EQ g` with a `B.NE h`. Under the D.a
 1312 and earlier text, the result could be consistent with executing `B.NE g` or `B.EQ h` instead, and thus the test
 1313 is allowed. Under the E.a-G.a text, the test has ‘constrained unpredictable’ behaviour. Under the G.b
 1314 and later text, the test has well-defined behaviour, but is now forbidden.

1315 To avoid this unfortunate confusion, and any possible constrained unpredictable behaviours due to it, our
 1316 examples will be restricted to modifying only NOPs and *unconditional* branches.

1317 **Synchronising branches** The Arm architecture does not give branch instructions any instruction synchronisation effects. Instead, the architecture relies on explicit synchronisation instructions (see §3.6). This is in contrast to other architectures. For example, x86 does not require any explicit cache maintenance or pipeline flushing when jumping to newly-modified code.

3.4 Coherence

Data writes and reads are coherent, in Arm and in other major architectures: in any execution, for each address, the reads of each hardware thread must see a subsequence of the total *coherence order* of all writes to that address (see §2.1.2). The plain-data CoRR1 test (Figure 2.5, p.21) illustrates one case of this: it is forbidden for a thread to read a new write of x and then the initial state for x .

Instruction fetches are not necessarily coherent: an instruction fetch may be inconsistent with a program-order-previous fetch, and the data and instruction streams can become out of sync with each other. However, they are not completely incoherent and still must respect some properties, giving rise to three new forms of coherence:

- ▷ Instruction-to-Instruction Coherence: whether fetches of the same location must observe writes to the same location coherently.
- ▷ Data-to-Instruction Coherence: whether fetches and then reads of the same location must observe writes to the same location coherently.
- ▷ Instruction-to-Data Coherence: whether reads and then fetches of the same location must observe writes to the same location coherently.

These new kinds of coherence describe the relationship between the instruction ‘stream’ with the instruction and data caches.

3.4.1 Instruction-to-Instruction coherence

Arm explicitly do not guarantee any consistency between fetches of the same location: fetching an instruction does not mean that a later fetch of that same location will not see an older instruction [66, B2.4.4]. This is illustrated by the CoFF test (Figure 3.4), which is a variant of the CoRR1 test (Figure 2.5, p.21) test for coherence discussed earlier, but where the explicit reads of the CoRR shape are replaced by the implicit reads of fetching the instructions.

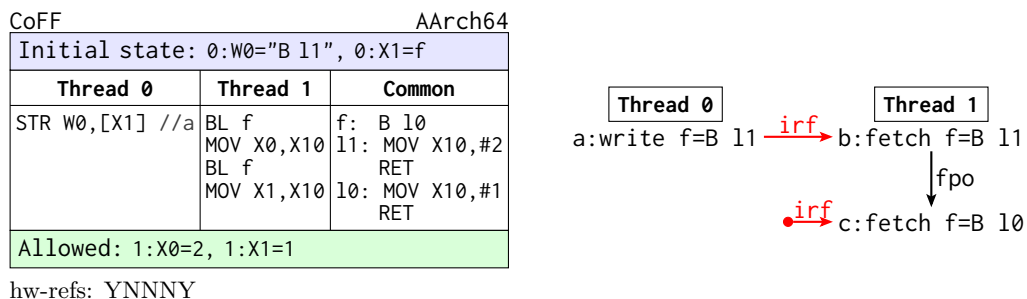


Figure 3.4: Code listing and execution diagram for CoFF.

Here, Thread 1 makes two calls to address f (recall BL is the branch-and-link ‘call’ instruction), while Thread 0 overwrites the instruction at that address with the opcode for the instruction B l1 (a branch to the location labelled l1). Here, and in future tests, we assume some common library code consisting of a function at address f , which always has the same shape: a branch that might be overwritten, which selects a block that writes a value to register X10 before returning. This is sometimes duplicated at different addresses ($f1$, $f2$, ...) or extended to g , with three cases. We sometimes elide the common code.

The interesting potential execution of this test is the one in which the first call to f fetches and executes the newly-written B l1, before the second call fetches and executes the original B l0. The execution shown in Figure 3.4 is the well-formed candidate execution consistent with the final state of the test. Candidate executions for self-modifying tests are similar to those of previous axiomatic models, but augmented with new fetch events, one per instruction, and new edges relating those events.

1355 As in Chapter 2, we use `po` and `rf` edges for the program-order and reads-from relations, together with
 1356 new relations:

- 1357 ▷ `fe` (fetch-to-execute), which relates the fetch event of an instruction to all the execution events
 1358 (memory writes, reads, and/or barriers) of the instruction;
- 1359 ▷ `irf` (instruction-read-from), relating a write to all fetches that read from it (analogous to reads-from,
 1360 `rf`); and
- 1361 ▷ `fpo` (fetch-program-order), relating fetches of instructions that are in program order (analogous to
 1362 program order, `po`).

1363 As usual, edges from the initial state are shown as originating from a small circle, for example, the `irf`
 1364 edge for event `c` in Figure 3.4. We discuss these new candidates in more detail later (Chapter 5).

1365 Since we do not modify the code of most locations, or perform any cache maintenance operations over
 1366 those locations, we usually omit the fetch events for the instructions at those locations. Instead, we show
 1367 only the subgraph of interesting events, as in the CoFF execution diagram in Figure 3.4.

1368 For Arm, this execution is both architecturally allowed and experimentally observed. This is shown in
 1369 the test listing in Figure 3.4 in the line underneath the final state beginning with `hw-refs`. This line is a
 1370 condensed table, where each column represents one hardware device and the entry represents whether it
 1371 was observed on that device (Y), not observed on that device (N), or whether there are no results for that
 1372 device (indicated by `-`). The final `hw-refs` line from CoFF (Figure 3.4, p.46), annotated with the names
 1373 of the devices (see §6.3 for a more detailed discussion of the hardware testing) is as follows:

	h955-a53	openq820	h955-a57	nexus9	s905
	N	Y	Y	N	N

1375 Where the devices are:

	h955-a53	Qualcomm Snapdragon 810 (cluster of 4x Arm Cortex A53)
	openq820	Qualcomm Snapdragon 820 (4x Qualcomm Kryo cores)
1376	h955-a57	Qualcomm Snapdragon 810 (cluster of 4x Arm Cortex A57)
	nexus9	NVIDIA Tegra K1 (with 2x NVIDIA Denver cores)
	s905	Amlogic 905 (with 4x Arm Cortex A53 cores)

1377 3.4.2 Data-to-Instruction coherence

1378 Fetching from a particular write does imply that program-order-later reads of the same address will see
 1379 that write, or something newer. This is a *data-to-instruction* coherence property, illustrated by CoFR
 1380 (Figure 3.5). Here, if Thread 1 happens to fetch the newly-written B 11 at `f` (in the ‘Common’ function
 1381 code), then a data read of `f` cannot see the original B 10 instruction (it can only read the new B 11).

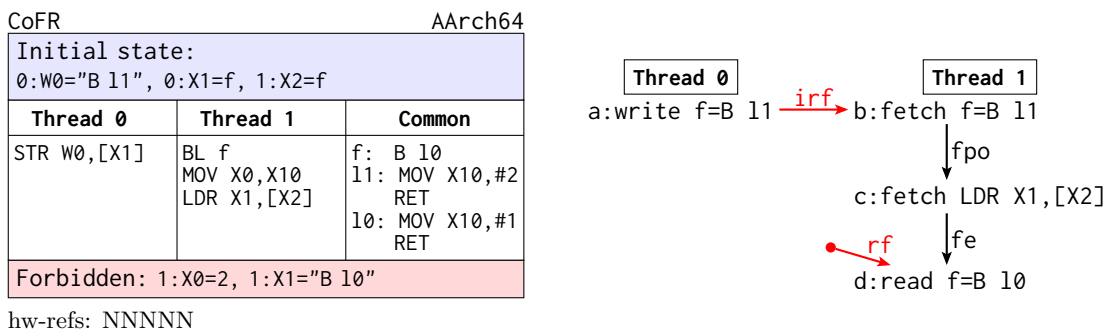


Figure 3.5: Code listing and execution diagram for CoFR.

1382 This ordering guarantee was not clear in the Arm prose specification at the time of this work [66, 77, 76],
 1383 but the architectural intent that emerged during discussion with Arm is that the given execution should
 1384 be forbidden. This architectural decision was motivated by microarchitectural design: (1) instructions
 1385 decode in order (so the fetch `b` must occur before the read `d`), and (2) fetches that miss in the instruction
 1386 cache must read from the coherent data storage system, so the instruction cache cannot be ahead of the

1387 available data. This ensures that observing a write with an instruction fetch implies that all threads are
 1388 now guaranteed to read from that write (or another coherence-after it).

1389 This test represents the most fundamental kind of data-to-instruction coherence: that data must become
 1390 visible to the coherent data side before instruction accesses. However, it alone gives no guarantee when the
 1391 instruction accesses are guaranteed to see it. We shall see later (§3.6) that instruction cache maintenance
 1392 will generally be required to guarantee future instruction fetches read from coherence-latest data writes,
 1393 but that the hardware may announce that it provides a stronger kind of data-to-instruction coherence
 1394 guarantee rendering such cache maintenance unnecessary (§3.14).

1395 3.4.3 Instruction-to-Data coherence

1396 In the other direction, reading from a particular write to some location does *not* imply that later fetches of
 1397 that location will see that write (or a coherence successor), as in the following CoRF+ctrl-isb (Figure 3.6).

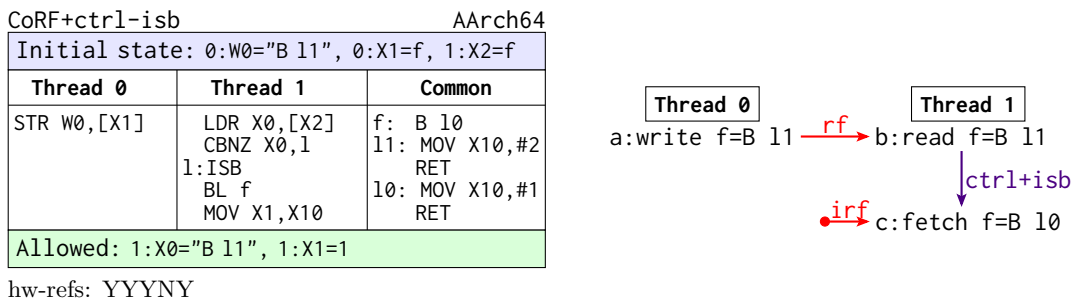


Figure 3.6: Code listing and execution diagram for CoRF+ctrl-isb.

1398 Here Thread 1 has a control dependency (the CBNZ conditional branch, dependent on the value read by its
 1399 load) and an instruction synchronisation barrier (ISB), abbreviated to ctrl+isb, between its load and
 1400 the fetch from f. If the latter were a data load, this would ensure the two loads are satisfied in order.
 1401 This was also not explicit in the prose [66, 77, 76], but it is what one would expect, and it is observed in
 1402 practice. Microarchitecturally, it is easily explained by an out-of-date entry for f in the instruction cache
 1403 of Thread 1: if Thread 1 had previously fetched f (perhaps speculatively), and that instruction cache
 1404 entry has not since been evicted or explicitly invalidated, then this fetch of f will simply read the old
 1405 value from the instruction cache without going out to data memory. The ISB ensures that f is freshly
 1406 fetched, but does not ensure that Thread 1's instruction cache is up-to-date with respect to data memory.

1407 However, even if the instruction cache is empty (e.g. by manually clearing it with appropriate cache
 1408 maintenance instructions, see §3.10 and the SM.F+ic test (Figure 3.19, p.55)) the test may still be
 1409 observed as the instruction fetches and instruction cache fills need not read-from the coherence-latest
 1410 write.

1411 Software must then use cache maintenance operations to achieve such guarantees (§3.6). However,
 1412 much like with data-to-instruction coherence, the hardware may announce that it provides a kind of
 1413 instruction-to-data coherence guarantee, rendering data cache maintenance unnecessary (§3.14).

3.5 Cross-thread synchronisation

We now consider modifying code that can be fetched by other threads, by considering variants of the standard message-passing shape `MP+pos` (Figure 2.1, p.18). Here, we replace one or both of the reads by fetches, and ask what synchronisation is required to ensure that the relaxed outcome is forbidden. Consider first an MP variant where the first write is of a new instruction, and the second is just a simple data memory flag, with some thread-local ordering ordering the writes on the left-hand thread, and ordering the read to the fetch on the right-hand side. We call this test `MP.RF+dmb+ctrl-isb` (Figure 3.7).

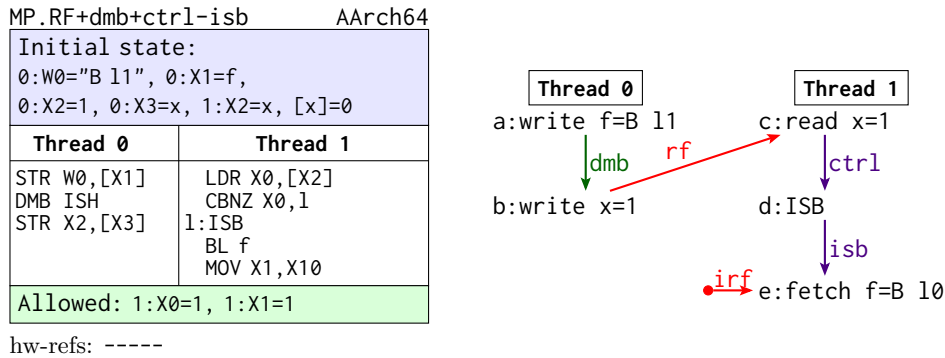


Figure 3.7: Code listing and execution diagram for `MP.RF+dmb+ctrl-isb`.

This test includes sufficient synchronisation on each thread to enforce thread-local ordering of data accesses: the DMB in Thread 0 ensures the writes a and b propagate to memory in program order, and the control dependency into an ISB on Thread 1 ensures the read c and the fetch e happen in program order. However, as we saw in §3.2, this is not enough to synchronise concurrent modification and execution of code in Arm-A. Thread 0 needs to perform the entire cache synchronization sequence (§3.2), not just a DMB, to forbid this outcome. Adding that full cache synchronization sequence gives test `MP.RF+cachesync+ctrl-isb` (Figure 3.11, p.51), described in more detail later (§3.6.2).

Synchronisation with memory by fetching Another variant of this MP-shape test, where the message passing itself is done using modification of code, gives a much stronger guarantee. This can be seen in `MP.FR+dmb+fpo-fe` (Figure 3.8), in which Thread 0 writes some data (to x) and then writes to the code concurrently being executed by Thread 1, as a kind of message pass. If Thread 1 fetches the new instruction written by Thread 0, then Thread 1 must also see the new value of x.

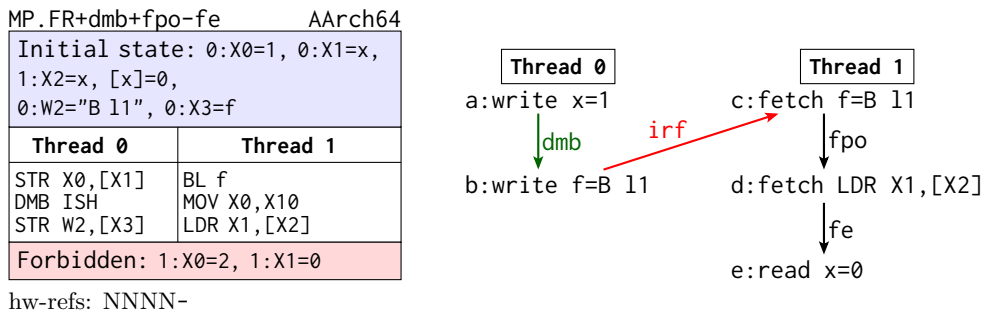


Figure 3.8: Code listing and execution diagram for `MP.FR+dmb+fpo-fe`.

This was not clear from the architectural prose at the time of the work, but this outcome is intended to be architecturally forbidden. This is for similar reasons as the previous `CoFR` test (Figure 3.5, p.47): since Thread 1 fetched the updated value for f, the value must have reached at least the data caches (since that is where the instruction cache reads from), and therefore multi-copy atomicity guarantees that a normal load instruction will observe it.

3.6 Cache maintenance

As we have seen, instruction fetches satisfy few guarantees, so explicit synchronisation must be performed when modifying the instruction stream to ensure correct execution of the new instructions.

Test SM (Figure 3.9) shows the simplest self-modifying code case: without additional synchronisation, a write to program memory can be ignored by a program-order-later fetch.

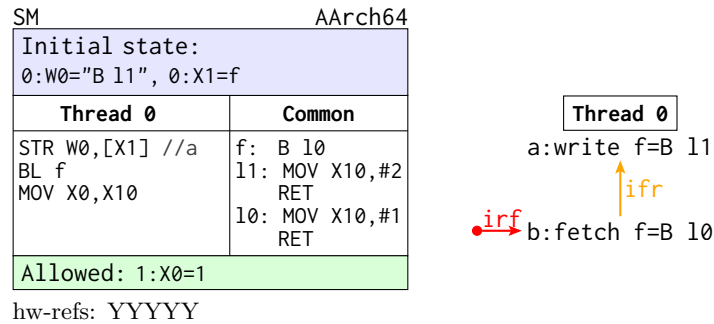


Figure 3.9: Code listing and execution diagram for SM.

In this execution, the fetch b, fetching the instruction at f, fetches a value from a write coherence-before a, even though b is the fetch of an instruction program-order after a. We illustrate this with an *instruction from-reads* (ifr) edge. This is a derived relation, analogous to the usual *from-reads* (fr) relation, that relates each fetch to all writes that are coherence-after the write it read from; it is defined as $ifr = irf^{-1};co$. If the fetch were a data read, this would be a forbidden coherence shape (CoWR). As it is, it is architecturally allowed, as described explicitly by Arm [66, B2.4.4], and it is experimentally observed on all devices we have tested. Microarchitecturally, there are a number of possible explanations, each of which are sufficient to explain the test: fetching b out-of-order with respect to a, fetching b from a stale entry in the instruction cache, or fetching b from memory after a has propagated but before it reaches the point the instruction fetch will see it. We will see that to forbid this test, and guarantee fetching the new instruction, one needs to account for all of those possibilities.

3.6.1 Synchronisation on a single thread

As we described earlier (§3.2), the Arm architecture provides cache maintenance instructions to synchronise the instruction and data streams: the DC data-cache clean instruction, and the IC instruction-cache invalidate instruction. To forbid the relaxed outcome of SM, by forcing a fetch of the modified code, the specified sequence of cache maintenance instructions must be inserted, with an ISB.

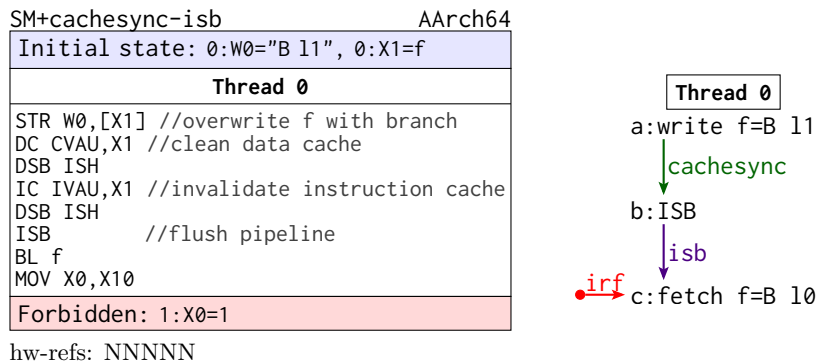


Figure 3.10: Code listing and execution diagram for SM+cachesync-isb.

Now, the outcome is forbidden. The cache synchronisation sequence DC CVAU; DSB ISH; IC IVAU; DSB ISH (which we abbreviate to a single *cachesync* edge) ensures that by the time the ISB executes, the instruction and data memory have been made coherent with each other for f. The ISB then ensures the final fetch of

1462 f is ordered after this sequence. The microarchitectural intuition for this sequence was in §3.2. Our §4.1
 1463 microarchitecturally-flavoured operational model will describe the semantics of this sequence using that
 1464 microarchitectural intuition in a way that gives precise and well-defined semantics to each instruction
 1465 individually, such that their composition results in the correct system-wide synchronisation. This will be
 1466 discussed in much more detail later (Chapter 4).

1467 3.6.2 Broadcast cache maintenance

1468 The hardware threads writing new instructions, performing the necessary cache maintenance, and finally
 1469 fetching the new instructions, may all be distinct. So long as the sequence in its entirety has been
 1470 performed by the time the fetch happens, then the instruction stream will have been made consistent
 1471 with the data stream for that address.

1472 The simplest example of this is in MP.RF+cachesync+ctrl-isb (Figure 3.11), where the ‘producer’ thread
 1473 (Thread 0) writes the new instructions, and performs all the cache maintenance, before writing a flag
 1474 informing the ‘consumer’ thread (Thread 1) that the instructions are ready to be fetched. Although the
 1475 cache maintenance happened on a different thread to the one that will try fetch the new instructions,
 1476 their effect is enforced system wide; the consumer needs only to flush its own pipeline (with an ISB)
 1477 to be guaranteed to see the new instructions.

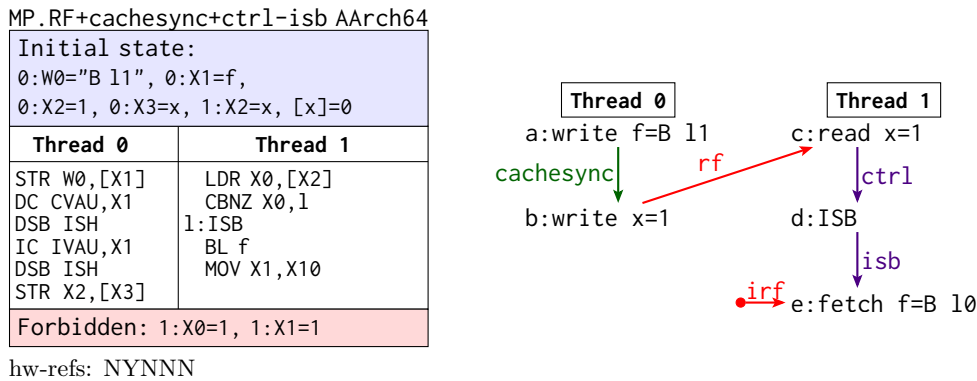


Figure 3.11: Code listing and execution diagram for MP.RF+cachesync+ctrl-isb.

1478 Note the positive observation on this test, despite being forbidden. We return to this in §6.4.1.

1479 **In-order fetches** One can make both writes of the MP shape be of new instructions. This idiom is quite
 1480 common in practice; this was how Chrome’s WebAssembly JIT synchronised its updates to modified code,
 1481 up until the code was redesigned to use Arm’s FEAT_BTI (branch-target-identification) feature [78, 79].
 1482 Without the full cache synchronisation sequence on Thread 0, this is allowed as in MP.FF+dmb+fpo
 1483 (Figure 3.12). Inserting the full cache maintenance sequence on the producer thread forbids the outcome,
 1484 see the MP.FF+cachesync+fpo test (Figure 3.13, p.52).

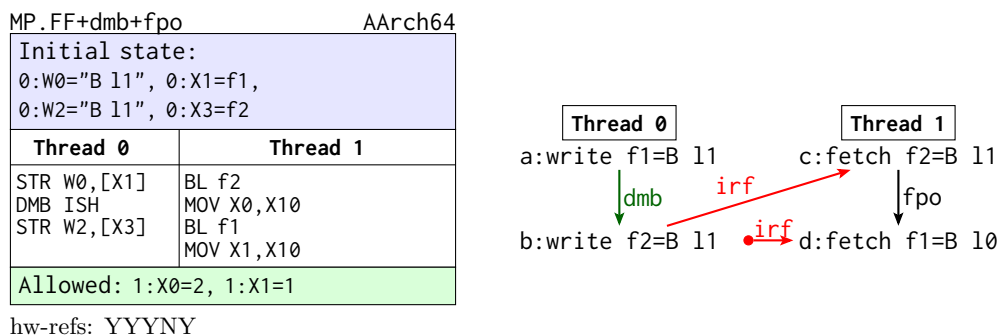


Figure 3.12: Code listing and execution diagram for MP.FF+dmb+fpo.

MP.FF+cachesync+fpo AArch64

Initial state: 0:W0="B 11", 0:X1=f1, 0:W2="B 11", 0:X3=f2	
Thread 0	Thread 1
STR W0,[X1] DC CVAU,X1 DSB ISH IC IVAU,X1 DSB ISH STR W2,[X3]	BL f2 MOV X0,X10 BL f1 MOV X1,X10
Forbidden: 1:X0=2, 1:X1=1	

hw-refs: NNNNN

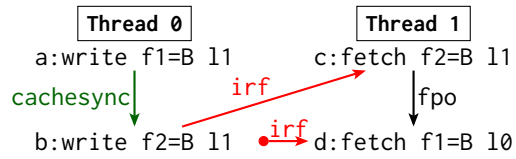


Figure 3.13: Code listing and execution diagram for MP.FF+cachesync+fpo.

1485 This may be surprising at first sight, as there is no synchronisation on the right-hand side (Thread 1), but
 1486 the architectural intent is for fetches to appear to be satisfied *in-order*.

1487 Microarchitecturally, that could be ensured in two ways: either by actually fetching in-order, or by making
 1488 the IC instruction not only invalidate all the instruction caches (for this address) but also clean any core's
 1489 pre-fetch buffer stale entries (for this address). Architecturally, this was not clear in the prose at the time of
 1490 the work, but, concurrent with this work, Arm were independently strengthening their definition to
 1491 guarantee this ordering.

1492 **Software thread migration** The cache maintenance sequence need not be contiguous, or even all on a
 1493 single thread. In general, it may be split up with many instructions between, and be over multiple threads.
 1494 This can be seen in the ISA2.F+dc-dmb+dsb-ic-dsb+ctrl-isb test (Figure 3.14), where Thread 0 performs
 1495 a write to f and then only a DC before synchronizing with Thread 1, which performs the IC, while Thread 2
 1496 observes the modified code. This can happen in practice when a software thread is migrated between
 1497 hardware threads at runtime, by a hypervisor or OS. Thread 0 and Thread 1 may just represent the
 1498 runtime scheduling of a single-threaded process, beginning execution on hardware Thread 0 but migrated
 1499 to hardware Thread 1 between the DC and IC instructions. In the graph, the dcsync and icsync represent
 1500 the DC and IC combinations with their surrounding barriers. The DC does not need a barrier preceding it,
 1501 because it is ordered w.r.t. the preceding store to the same cache line.

ISA2.F+dc-dmb+dsb-ic-dsb+ctrl-isb AArch64

Initial state: 0:W0="B 11", 0:X1=f, 0:X2=1, 0:X3=x, [x]=0, 1:X4=f, 1:X1=x, 1:X2=1, 1:X3=y, [y]=0, 2:X2=y		
Thread 0	Thread 1	Thread 2
STR W0,[X1] DC CVAU, X1 DMB SY STR X2,[X3]	LDR X0,[X1] DSB ISH IC IVAU, X4 DSB ISH STR X2,[X3]	LDR X0,[X2] CBZ X0,1 1:ISB BL f MOV X1,X10
Forbidden: 1:X0=1, 1:X1=1		

hw-refs: NN---

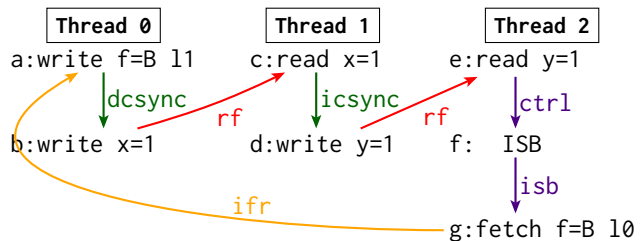


Figure 3.14: Code listing and execution diagram for ISA2.F+dc-dmb+dsb-ic-dsb+ctrl-isb.

1502 This works because the IC IVAU is broadcast to all threads [66, B2.2.5p3]. Therefore the IC happening
 1503 on a different thread to the DC does not break the sequence, so long as there is ordering between the IC
 1504 and DC. Additionally, the DC need not happen on the same thread as the initial store, so long as the DC is
 1505 ordered after the store.

1506 The migration and context-switching code needs only contain a DSB and a context-synchronising operation
 1507 (such as an ISB, although usually this is performed implicitly by the exception return mechanism itself)
 1508 to ensure sufficient synchronisation exists for the sequence to be migrated at any point.

3.7 Dependencies

Reads, including implicit reads due to an instruction fetch, must have their address become known before the value can be used. This is a general principle Arm have, that values from reads generally cannot be observably speculated. For instruction fetches, this address is the program counter.

This means that computations which are used in the calculation of that address give rise to *dependencies* in the program. Sometimes these dependencies are hard and must be preserved, and other times, not.

3.7.1 Address dependencies

When the destination of a branch is computed, e.g. with the BR (branch-register) or BLR (branch-and-link-register) instructions, then the instruction fetch of the target cannot go ahead until after the address is resolved. This can be seen in the `MP.RF+cachesync+addr` test (Figure 3.15), where the target of the branch is dependent on the value of register X2 which comes from the earlier load of x.

MP.RF+cachesync+addr AArch64

Initial state:	
0:W0="B l1", 0:X1=f, 0:X2=1, 0:X3=x, 1:X2=x, [x]=0	
Thread 0	Thread 1
STR W0,[X1] DC CVAU,X1 DSB ISH IC IVAU,X1 DSB ISH STR X2,[X3]	LDR X0,[X2] EOR X2,X0,X0 ADD X2,X2,f BLR X2 MOV X1,X10
Forbid?: 1:X0=1, 1:X1=1	

hw-refs: -----

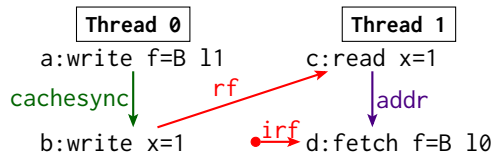


Figure 3.15: Code listing and execution diagram for `MP.RF+cachesync+addr`.

3.7.2 Control dependencies

For branches where the destination is known, but where it is not yet known if the branch will be taken, then it is permitted for the instruction to be fetched and executed speculatively.

MP.RF+cachesync+ctrl AArch64

Initial state:	
0:W0="B l1", 0:X1=f, 0:X2=1, 0:X3=x, 1:X2=x, [x]=0	
Thread 0	Thread 1
STR W0,[X1] DC CVAU,X1 DSB ISH IC IVAU,X1 DSB ISH STR X2,[X3]	LDR X0,[X2] CBNZ X0,1 1: BL f MOV X1,X10
Allowed: 1:X0=1, 1:X1=1	

hw-refs: YYYYYY

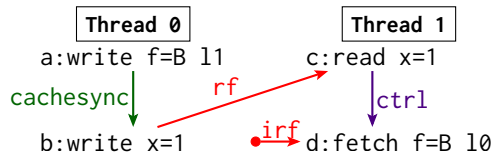


Figure 3.16: Code listing and execution diagram for `MP.RF+cachesync+ctrl`.

3.8 Multi-Copy Atomicity

For data accesses, the question of whether they are *multi-copy atomic* is a crucial one in relaxed architectures. IBM POWER, ARMv7, and pre-2018 ARMv8-A are *non-multi-copy atomic*: two writes to different addresses could become visible to distinct other threads in different orders. Post-2018 ARMv8-A, Armv9-A, and RISC-V are all multi-copy atomic (or ‘other multi-copy-atomic’ in Arm terminology) [7, 6, 66]: the programmer can assume there is a single shared memory, with all data-access relaxed-memory effects due to thread-local out-of-order execution.

The lack of any fetch atomicity guarantee for most instructions (§3.3), and the lack of coherent fetches for the others (§3.4), means the question of multi-copy atomicity for instruction fetching is not particularly interesting. Tests are either trivially forbidden (by data-to-instruction coherence, as in test WRC.F.RR+po+dmb (Figure 3.17)) or are allowed, but only the full cache synchronisation sequence provides enough guarantees to forbid it, and this sequence ensures all cores will share the same consistent view of memory.

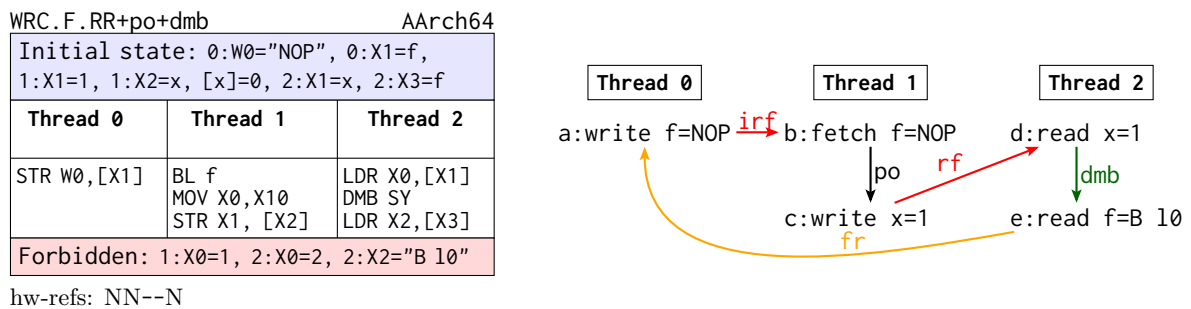


Figure 3.17: Code listing and execution diagram for WRC.F.RR+po+dmb.

3.9 More on instruction caches

Test CoFF (Figure 3.4, p.46) showed that fetches can see ‘old’ writes. In principle, there is no limit to the number of distinct values within the instruction cache: there could be many values for a single location cached in the instruction memory for each core, even if the data cache has been cleaned. The MP.RFF+dc-dsb+ctrl-isb-isb test (Figure 3.18) illustrates this, with Thread 0 writing two distinct new opcodes for g, and Thread 1 able to see all three (both of the new, and the initial) values for g. If the instruction cache could hold at most one value for each location, then after a DC an instruction fetch could read at most two values: one from that and one from data memory. Although it is unlikely that hardware would cache multiple values in the instruction cache, the desire for the simpler and weaker option means the architectural intent is to allow it, and we follow that in our models.

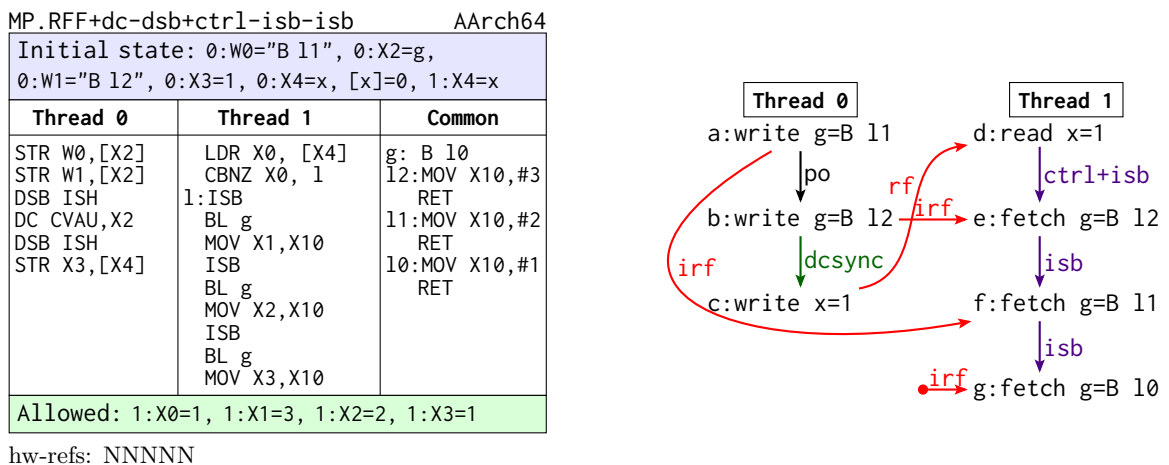


Figure 3.18: Code listing and execution diagram for MP.RFF+dc-dsb+ctrl-isb-isb.

3.10 Points of Unification and Coherency

Although instruction caches behave incoherently, at some point in the memory hierarchy all the caches unify, and all agents agree on the value at that point. Arm define multiple such points:

- ▷ The Point of Unification (for each shareability domain, see below) is where the data and instruction caches unify, and an instruction cache fill is guaranteed to see writes beyond.
- ▷ The Point of Coherency (for the whole system) is where all agents (CPUs, devices, etc.) memory accesses unify, and are all guaranteed to see the same writes beyond that point.

Cleaning the data cache, with the DC instruction, forces previous writes to become visible to instruction fetch, but does not restrict the set of values that could be in the instruction cache. It does this by pushing the writes past the Point of Unification (the point where the instruction and data caches become unified).

There may be multiple Points of Unification, one for each shareability domain. For example, one for each individual core, where its own instruction and data memory become unified, and one for each cluster of CPUs where all the caches eventually unify within that cluster. When a cache maintenance operation is performed by VA to ‘the’ Point of Unification, the VA is translated and an entry in the translation tables marks the location as either non-shareable or inner-shareable (see §7.3.2). This determines which Point of Unification the cache maintenance operation should be performed to.

Fetching a value implies that its write has reached at least that core’s PoU, but not necessarily the PoU of any wider domain, even if the write originated from a different core. Consider test SM.F+ic (Figure 3.19).

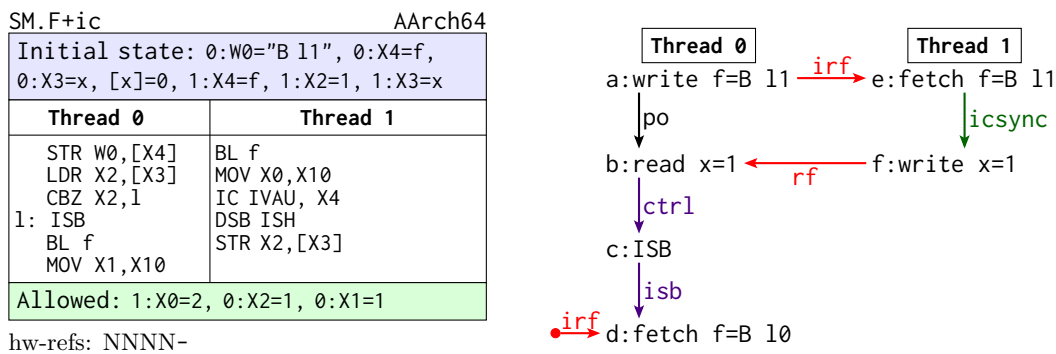


Figure 3.19: Code listing and execution diagram for SM.F+ic.

In SM.F+ic, Thread 0 modifies f, and Thread 1 fetches the new value and performs just an IC and DSB, before signalling Thread 0 which also fetches f. The IC (without its sibling DC) is not strong enough to ensure that the write is pulled into the instruction cache of Thread 0.

We have not so far observed this in practice. However, there is a mechanism which gives a compelling argument to permit this behaviour. The Points of Unification of systems are not necessarily hierarchical: the write may have passed the Point of Unification for Thread 1, but not the shared Point of Unification for both threads. In other words, the write might reach Thread 1’s instruction cache without being pushed down from Thread 0’s data cache. Microarchitecturally this can be explained by *direct data intervention* (DDI), an optimisation allowing cache lines to be migrated directly from one thread’s (data) cache to another [80]. The line could be migrated from Thread 0 to Thread 1, and pushed past Thread 1’s Point of Unification making it visible to Thread 1’s instruction memory, without ever making it visible to Thread 0’s own instruction memory. The lack of coherence between instruction and data caches would make this observable in theory, even on multi-copy atomic machines, if they implemented pre-PoU DDI. Therefore, the architectural intent is to allow this behaviour.

With insufficient synchronisation of the data caches, there is theoretically no limit to how far back in time the fetches could read from. Recall that in the MP.RF+dmb+ctrl-isb test (Figure 3.7, p.49), the full cachesync sequence was required to forbid the ‘bad’ behaviour. Test FOW (Figure 3.20, p.56) is similar to that MP-shaped test, but writes two new values to the data consecutively rather than one, and has two threads reading the flag before fetching that address. Here, both threads can see the updated flag, but can execute different instructions on the instruction fetch of g, even after invalidating the instruction cache.

FOW			AArch64
Initial state: 0:W0="B 11", 0:X2=g, 0:W1="B 12", 0:X3=1, 0:X4=x, [x]=0, 1:X4=x, 2:X4=x			
Thread 0	Thread 1	Thread 2	Common
STR W0,[X2] STR W1,[X2] DSB ISH IC IVAU, X2 DSB ISH STR X3,[X4]	LDR X0, [X4] CBNZ X0, 1a 1a: ISB BL g MOV X1,X10	LDR X0, [X4] CBNZ X0, 1b 1b: ISB BL g MOV X1,X10	g: B 10 12: MOV X10, #3 RET 11: MOV X10, #2 RET 10: MOV X10, #1 RET
Allowed: 1:X0=1, 1:X1=2, 2:X0=1, 2:X1=1			

hw-refs: NN--N

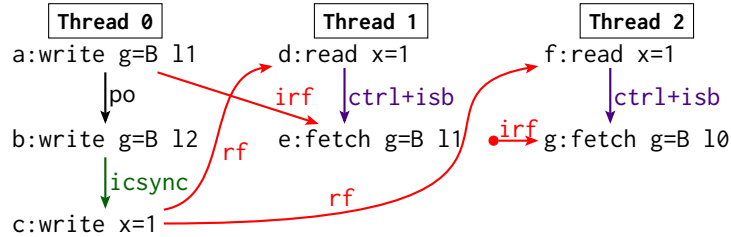


Figure 3.20: Code listing and execution diagram for FOW.

1584 This was not clear in the existing architecture text. It is a case where the architecture design is not very
 1585 constrained. On the one hand, it has not been observed, and it is thought unlikely that hardware will ever
 1586 exhibit this behaviour: it would require keeping multiple writes in the coherent part of the data caches,
 1587 before the *Point of Coherency*, which would require more complex cache coherence protocols, rather than
 1588 a single dirty line. On the other hand, there does not seem to be any benefit to software from forbidding
 1589 it. Therefore the architects are forced to make a decision. In this case, the more permissive model is also
 1590 the simpler one. It makes it easier for programmers to understand and to provides more flexibility for
 1591 future microarchitectural optimisations. Our models therefore allow the above behaviour.

1592 In theory, once a write passes the Point of Unification for the whole system (the point where all caches
 1593 eventually unify) then any writes coherence before that write cannot be seen at all by instruction fetches
 1594 any more, even without explicit DC instructions. We do not attempt to model this since a general notion
 1595 of a Point of Coherency is not required in the models, as it is only distinguished by device memory or
 1596 DMA, which we do not model here.

1597 3.10.1 Late unification

1598 There is a final question: can any Point of Unification be *after* the Point of Coherency? The architecture
 1599 says [81, D7.5.8, p.5777]:

Point of Coherency The point at which all agents that can access memory are guaranteed to see the same copy of a memory location [...]

1601 The question is whether the instruction-fetch-units of a system amount to separate *agents* in the system
 1602 domain. The definition of shareability domains implies not [81, B2.4.2, p.199]:

Shareability [...] Marking a memory location as shareable [...] requires hardware to ensure [...] the location is coherent for all agents in that domain. [...]

1603 Since the instruction fetch unit is within the same shareability domain, but are not coherent, they must
 1604 not be *agents* for the purpose of coherency. Therefore, writes passing the Point of Coherency would not
 1605 guarantee instruction fetches to see the writes coherently. We never observed such hardware in practice,
 1606 and believe it unlikely hardware would be designed in such a way that the PoC would be before a PoU,
 1607 and therefore leave it open to the architects to clarify.
 1608

1609 3.10.2 Promotion

Cache maintenance operations form a partial order, where if one cache operation is sufficient, then a stronger one is also sufficient. Assuming the PoU is before the PoC (see [Late unification](#)) that order is:

$$\text{DC CVAU} \leq \text{DC CVAC} \leq \text{DC CIVAC}$$

$$\text{DC IVAC} \leq \text{DC CIVAC}$$

$$\text{IC IVAU} \leq \text{IC IVAC}$$

$$\text{IC IVAU} \leq \text{IC IALLUIS}$$

1610 In litmus tests we will use the least operations in this order, typically DC CVAU and IC IVAU.

1611 In a program which uses one of these instructions, that instruction can be *promoted*: replaced with a
1612 stronger cache maintenance operation. Often software will want to use the least sufficient maintenance as
1613 they are typically the most efficient and give the best performance. However, sometimes operating systems
1614 and hypervisors will ‘trap’ cache maintenance operations to emulate or promote them automatically, either
1615 for virtualisation or as part of the resolution to CPU errata. In those cases, software must ensure it only
1616 promotes cache maintenance consistent with the above ordering.

1617 3.11 Cleans and invalidates are like reads and writes

1618 Recall that we have an asymmetry between the required synchronisation around DC instructions and
1619 IC instructions: IC instructions must have a preceding DSB to order with earlier accesses, whereas DC
1620 instructions do not necessarily need one; DC instructions are ordered by DMB with surrounding memory
1621 accesses, whereas an IC is not.

1622 This is because the clean of the DC is ordered much like a read. However, both the DC and IC are not
1623 guaranteed to have completed their effect until after the subsequent execution of a DSB instruction on the
1624 same thread [81, pp. 5790-5791], and an IC instruction always requires an DSB to order accesses before it
1625 [81, p. 5791].

1626 3.11.1 Cleans are similar to reads

1627 Microarchitecturally, cleans are non-destructive; they push the data further down the cache hierarchy,
1628 without causing the data to be lost. In hardware, these clean operations may be propagated around the
1629 system in much the same way reads are. This gives clean operations the same memory ordering constraints
1630 as data reads. This, in turn, means that DC C_VA_ instructions wait for program-order previous reads
1631 and writes (and other DCs) of the same location just as reads do (or really, within the same cache line
1632 of minimum size, see §3.12), and do not require any other explicit barriers or dependencies between
1633 them. Cleans may be speculated, but otherwise respect dependencies and fences, even with respect to
1634 surrounding non-same-cache-line accesses.

1635 3.11.2 IC invalidates are not quite like writes

1636 Invalidation is destructive: data that was once visible is potentially lost. Data cache invalidations
1637 behave somewhat like writes: they cannot be performed speculatively; and end up existing at some place
1638 within the global coherence order of that location, such that reads after the invalidation cannot read from
1639 writes from before it. IC invalidations behave slightly differently, with some extra requirements about
1640 in-order fetching (see test [MP.FF+dmb+fpo](#) (Figure 3.12, p.51)), and without constraining future *data*
1641 reads, and they do not respect dependencies or barriers other than DSB. This means that, in practice,
1642 every IC requires a DSB between it and any program-order earlier or later memory accesses, in order to
1643 synchronise with them.

1644 **3.11.3 DC and IC address speculation**

1645 Normal data load and store instructions (in Arm-A and in other relaxed architectures) respect *address*
 1646 *dependencies*: reads cannot be satisfied, and writes cannot be forwarded from or committed, until their
 1647 addresses are resolved from previous register writes (though those can still be out-of-order or speculative).
 1648 In other words, the architecture forbids programmer-visible value speculation of such addresses.

1649 For DC and IC instructions, which are loosely analogous to loads and stores from the specified addresses,
 1650 we similarly have to consider whether or not dependencies from the calculation of their addresses are
 1651 respected. Test [MP.R.RF+addr-cachesync+dmb+ctrl-isb](#) (Figure 3.21) illustrates this for DC. Thread 0
 1652 writes to *g* and performs the full cache synchronization sequence. However, the DC's address depends
 1653 on a detour through Thread 1 which writes an even newer instruction to *g*. Since the address of the DC
 1654 cannot be speculated, this address dependency must be preserved and so the final fetch of *g* after the
 1655 cache synchronization must observe the branch Thread 1 wrote.

1656 This was unclear in the prose at the time of this work, but Arm have since decided the architectural
 1657 intent is that it should be forbidden: addresses of cache maintenance instructions should not be visibly
 1658 value-speculated, and so these instructions must respect their address dependencies.

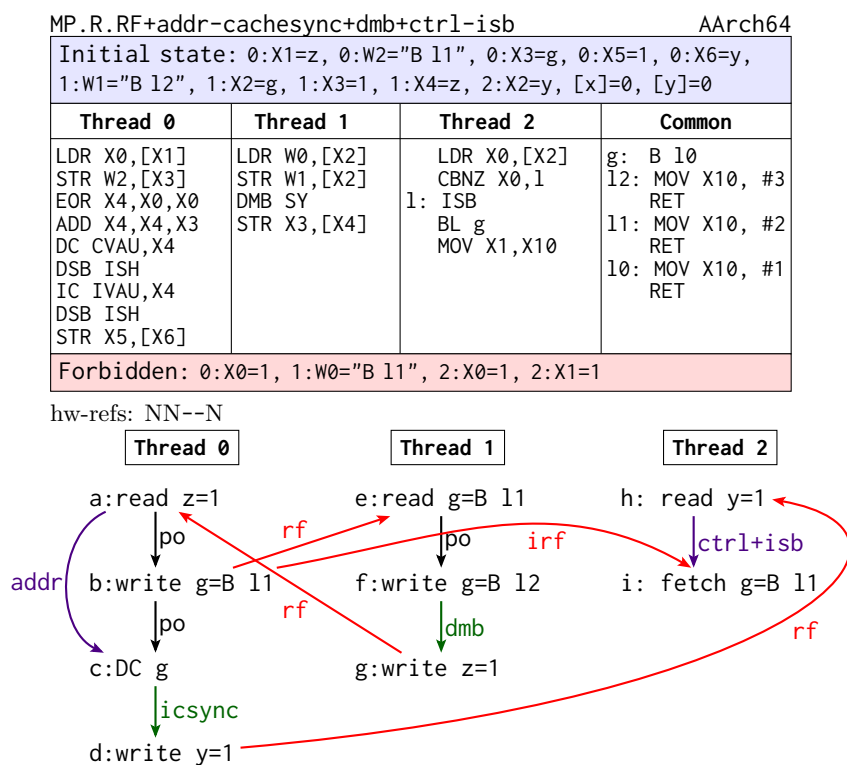


Figure 3.21: Code listing and execution diagram for MP.R.RF+addr-cachesync+dmb+ctrl-isb.

1659 **3.11.4 DC might be to same address**

1660 Data loads and stores can be ordered by the fact that they might access the same address [40, §12.5].
 1661 Arm made it clear in the architectural text that DC is ordered with respect to loads and stores with
 1662 addresses in the same cache line, while IC is not [66, D4.4.8]. We therefore have to ask whether DC is
 1663 subject to a might-access-same-address restriction in the same way as data loads and stores [40, §10.5].
 1664 The `MP.RRF+dmb+addr-cachesync-isb` test (Figure 3.22) below illustrates this, in which program-
 1665 order previous load/store addresses may not be determined when the DC executes. Arm clarified that
 1666 the architectural intent (which was not clear from the architectural text at the time of this work) is
 1667 that DC should be like loads in this respect too, with the aforementioned test architecturally allowed.
 1668 Microarchitecturally, the DC is not required to wait for those addresses to be determined before executing,
 1669 but if they end up being to the same address, the DC must be re-issued. Because the read `d` was not to the
 1670 same location, the DC need not be re-issued and so may have happened before the write `a` to `f`.

MP.RRF+dmb+addr-cachesync-isb		AArch64
Initial state: 0:W0="B l1", 0:X1=f, 0:X2=1, 0:X3=x, [x]=0, 1:X1=x, 1:X4=z, [z]=0, 1:X5=f		
Thread 0	Thread 1	
STR W0, [X1] DMB SY STR X2, [X3]	LDR X0, [X1] EOR X2, X0, X0 LDR X3, [X4, X2] DC CVAU, X5 DSB ISH IC IVAU, X5 DSB ISH ISB BL f MOV X6, X10	
Allowed: 1:X0=1, 1:X6=1		

hw-refs: N-N--

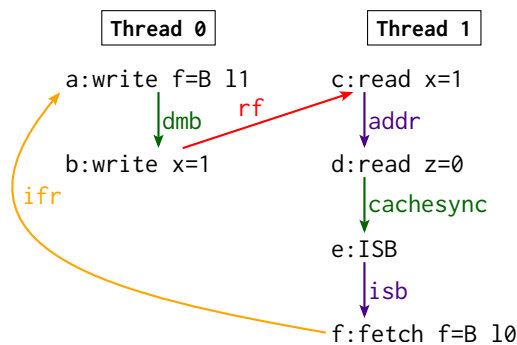


Figure 3.22: Code listing and execution diagram for `MP.RRF+dmb+addr-cachesync-isb`.

1671 **3.11.5 DC ordering with respect to other memory accesses**

1672 We saw that the DC instruction is ordered with program-order-previous stores to the same address. Normal
 1673 ‘data’ loads are additionally ordered with respect to other same-location accesses in the same thread. Here
 1674 we ask how far we can extend this to data cache maintenance operations.

1675 **po-previous loads** We extend this to cover all the natural thread-local same-address ordering constraints
 1676 as normal ‘data’ loads. For example, DCs are ordered with respect to program-order-earlier same-location
 1677 loads as in **CoRF+cachesync-isb** (Figure 3.23), and may be re-ordered with respect to program-order-later
 1678 same-location loads, as in **MP+dmb+addr-dc** (Figure 3.24).

1679 Note that these have not yet been confirmed with Arm architects; where the test final state has a question
 1680 mark, the stated results come from our models and await architectural decision.

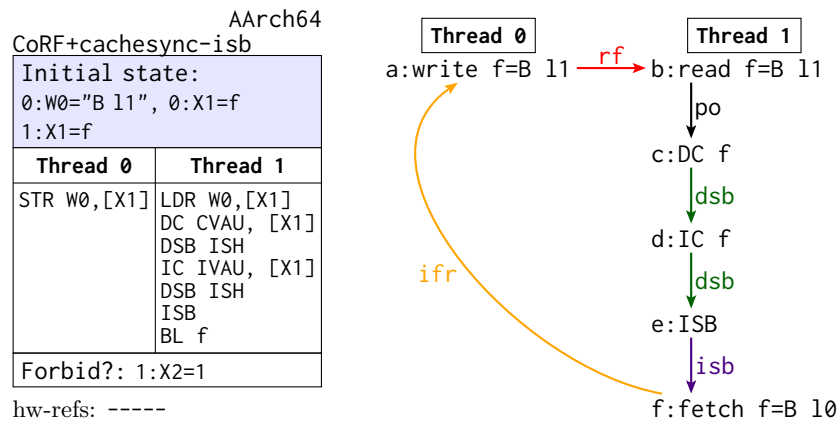


Figure 3.23: Code listing and execution diagram for CoRF+cachesync-isb.

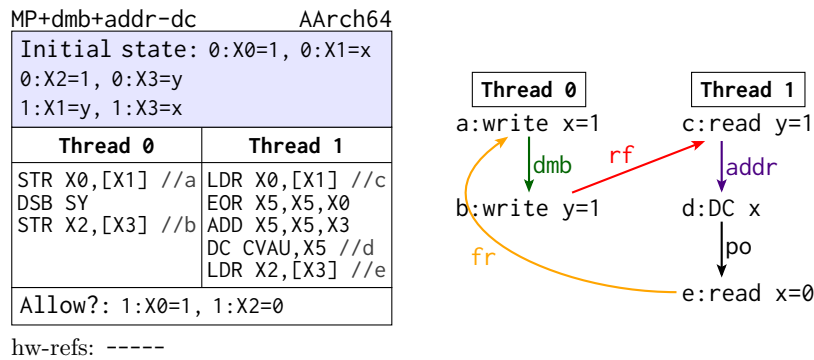


Figure 3.24: Code listing and execution diagram for MP+dmb+addr-dc.

3.12 Same-cache-line ordering

Arm-A has an architected *cache line of minimum size*. There are two cache line minimum sizes: one for the data caches, and one for the instruction caches. They are accessible as the DMinLine and IMinLine bitfields of the CTR_EL0 register, which encode \log_2 the number of (32-bit) words in the smallest cache-line size¹, for the data and instruction caches, respectively.

Accesses being within the same cache line does not impose additional ordering constraints [16], unless one of the accesses is a cache maintenance operation. For example, the SB+scls test (Figure 3.25), which is a variation of the classic store buffering example where the two locations are to the same cache line, is still allowed as the reads and writes of different locations (even within the same cache line) are not ordered.

In this test, X is an array of size $2^{2+DMinLine}$ bytes, and X is aligned on a cache boundary, therefore X and X+4 are 32-bit aligned addresses in the same (data) cache line of minimum size.

This is separate to concerns about mixed-size accesses, which we consider in §3.13, where a program writes to the same location with architected writes of different size.

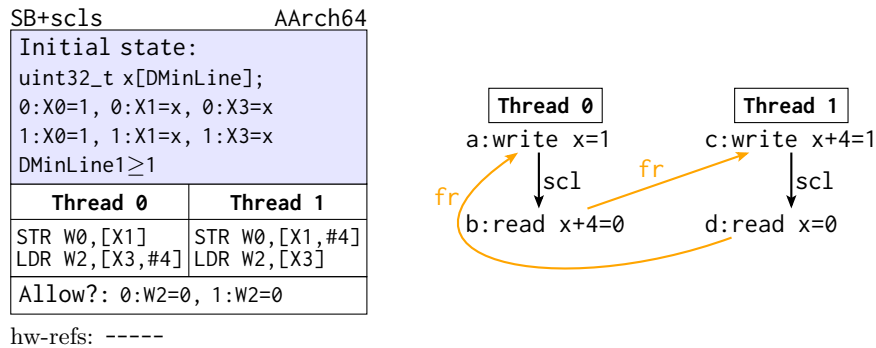


Figure 3.25: Code listing and execution diagram for SB+scls.

DC to same cache line Given two locations f and g in the same cache line of minimum size, performing the cache clearing sequence for one will also clear the other, as in SM+sclcachesync-isb (Figure 3.26)

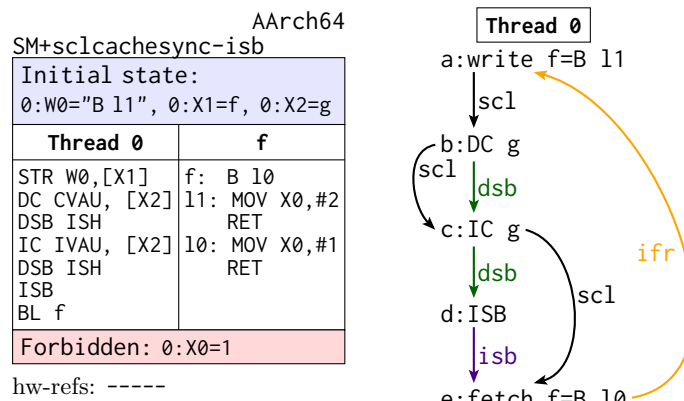


Figure 3.26: Code listing and execution diagram for SM+sclcachesync-isb.

¹Note that, while the encoding allows DMinLine and IMinLine to be zero, this assignment does not make much sense for hardware, and it is likely no implementation exists with either less than the size of the largest implemented single-copy atomic access size.

3.13 Mixed-size instruction fetching

In the tests so far we have always replaced a single instruction with another whole instruction, with a single write. However, it is easy to imagine code that replaces an instruction byte-by-byte, or perhaps even only replacing a single field in the instruction encoding.

It is clear that performing individual per-byte writes and then performing the full cache synchronization sequence, without concurrently attempting to fetch the location, should give the desired result without unpredictable behaviour.

For example, in the `SM8+sclcachesync-isb` test (Figure 3.27), a new 32-bit instruction is written byte-by-byte before performing a full cache synchronisation sequence on a single core. Here, it is not a *concurrent* modification of the location, as it is all on a single core and the sequence is complete before the fetch happens, and so the result is a well-defined forbidden outcome. This pattern can occur in practice, as code often gets loaded from some other memory by means of some memory copying code, which may copy bytes using instructions whose accesses are not naturally instruction-sized, before they are executed.

Note that the 32-bit opcode for B 11 differs from that of B 10 only in the last byte (at `f[0]` since instructions are always stored little-endian in Arm-A), so all combinations of the writes correspond to instructions which are in the set of modifiable instructions. One can also delete the final three `STRB` instructions (events b-d) from the test, and not affect the result (it is still forbidden).

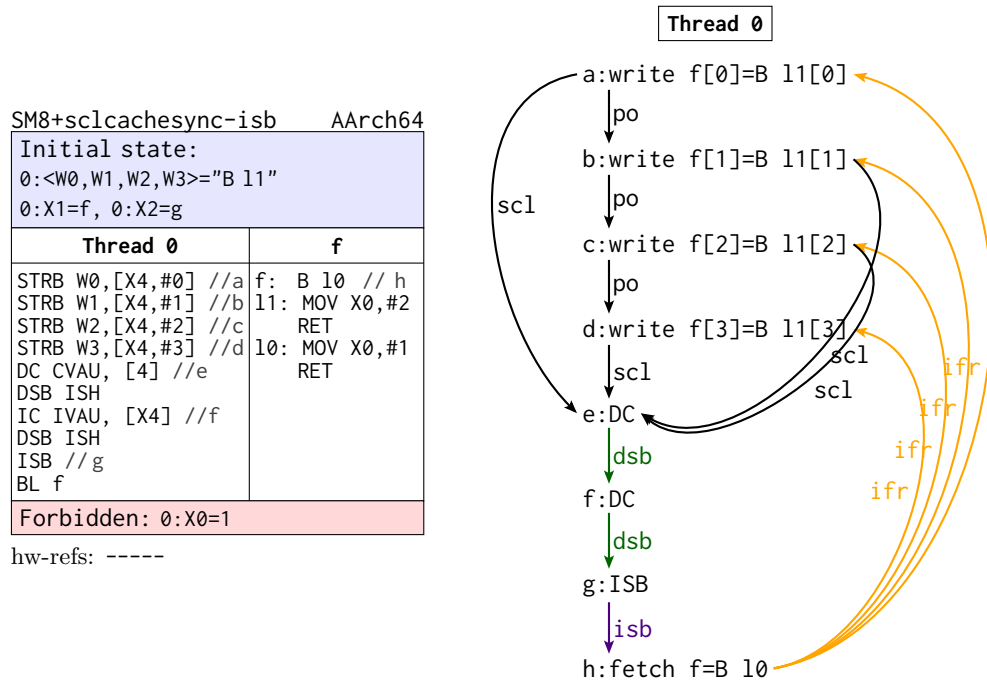


Figure 3.27: Code listing and execution diagram for `SM8+sclcachesync-isb`.

It is less clear in the architectural prose (even as of the most recent version, J.a [81]) what happens if one were to *concurrently* modify part of an instruction, either in a single thread without sufficient synchronisation as in `SM+mixed` (Figure 3.28, p.63), or across multiple threads as in `W+F+mixed` (Figure 3.29, p.63). We do not discuss this in detail, and we are not aware of any software patterns that rely on it. We leave this question open for the architects to resolve at a later time.

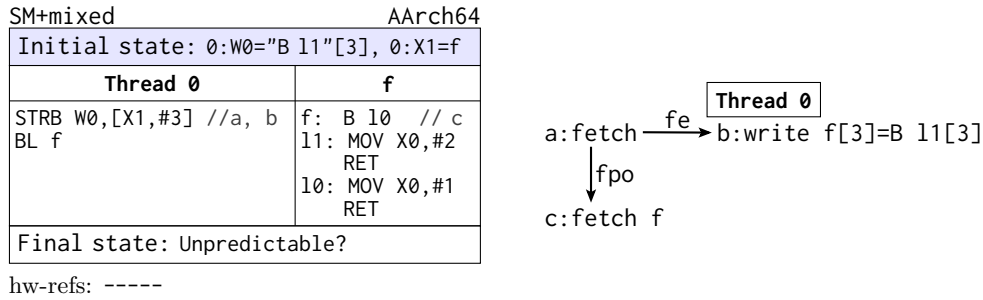


Figure 3.28: Code listing and execution diagram for SM+mixed.

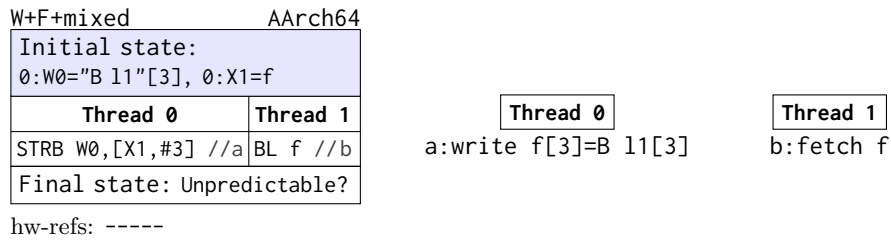


Figure 3.29: Code listing and execution diagram for W+F+mixed.

3.14 Cache type strengthening: IDC and DIC

Implementations may announce that they provide stronger guarantees through two fields in the cache type identification register (CTR_EL0). They are the IDC and DIC fields. The value of these fields then inform software whether each of the cache maintenance instructions are required.

IDC is related to instruction-to-data coherence, and requirements on data cache maintenance. DIC is related to data-to-instruction coherence, and the requirement for instruction cache maintenance. As the names suggest, these fields are related to the kinds of coherence introduced in Section 3.4.

If implementations choose to advertise that one or other of the cache maintenance operations are not required, then those cache maintenance instructions simply become hints or NOPs, so defensive cleans and invalidations will not be harmful to the program.

None of the devices we tested had either strengthening enabled.

3.14.1 IDC

When CTR_EL0.IDC is 1, the DC instruction is not required as part of the sequence [81, p. 201].

Point of Unification When the DC instruction is not required, it means that writes must reach the Point-of-Unification before being propagated to other threads. This means, under IDC=1, the earlier SM.F+ic test (Figure 3.19, p.55) is forbidden.

3.14.2 DIC

When CTR_EL0.DIC is 1, the IC instruction is not required as part of the sequence [81, p. 201].

In-order fetching Recall that instruction fetches must either happen in-order, or the IC instruction must touch the internal fetch queues of the individual threads (§3.5). When DIC=1, the IC instruction is not required, and this forces fetches to be satisfied from the instruction cache in the order they are fetched into the fetch queue. This is exactly how our operational model is expressed (which we shall see in Chapter 4).

3.15 Related Work

Explicit cache maintenance makes these tests, and the models presented in the next two chapters, quite different to the ‘user mode’ relaxed memory models discussed in [Chapter 2](#).

Previous work on verification, of operating systems, hypervisors, and JITs, has had to work with idealised models of the underlying hardware.

Myreen’s JIT compiler verification [65] models x86 icache behaviour with an abstract cache that can be arbitrarily updated, cleared on a `jmp`.

Cai, Shao, and Vaynberg produce a Hoare-style logic for certifying programs which contain self-modifying patterns [82], extending a version of *Concurrent Abstract Predicates* (CAP) [83] for generalised von-Neumann machines.

Goel et al.’s work on verification of x86 machine code programs [84, 85] includes a system step relation, based on their idealised x86 instruction models in ACL2. This model fetches instructions from memory, but avoids the complexity of caches and pipelines [86].

Lustig et al. describe a framework for concurrent models, with relaxed behaviours, for machine code x86 programs based on stages of hardware micro-operations [87]. They produce some models in this framework which include instruction fetching and the (data and TLB, not instruction) caches of a specific hardware implementation. These models explain behaviours seen based on knowledge of the underlying microarchitecture, but are not intended to be architectural models.

The verification of seL4 [55] included self-modifying patterns, but assumed the correctness of the required cache maintenance, without producing tight architectural models of the individual instructions.

CertiKOS [56, 57] verifies an assortment of safety and security properties (no code injection, no buffer overflows, no data races, and so on) for a custom-written kernel, with respect to an underlying concurrent, but not relaxed, x86 hardware machine model (‘x86mc’) without self-modifying code .

SeKVM [88] similarly verified a custom-written (in this case, for Arm) micro-kernel, with respect to an underlying concurrent, and somewhat relaxed, hardware model. This model is far less idealised than those used in earlier verification efforts (but still not an architectural definition by any means), such as those in the seL4 and CertiKOS projects. The *KCore* kernel itself does not require self-modifying code, the contextual refinement did not consider programs with concurrent or self-modifying code, and the underlying hardware model did not support data or instruction cache maintenance operations.

For architectural models which include cache maintenance, the closest is Raad et al.’s work on non-volatile memory. They model the required cache maintenance for persistent storage in ARMv8-A [89], as an extension to the ARMv8-A axiomatic model, and for Intel x86 [90] as an operational model.

There is also some work on address translation and TLB maintenance, which has a very similar flavour to cache maintenance. We explain the related work on TLBs in more detail later ([§8.10](#)).

During this work, Arm informally confirmed they would adopt the model (subject to necessary updates and changes of architectural intent) [67].

Independent work by Arm, after the conclusion of this work, further extended the herdttools suite of tools, models, and tests, for instruction fetching and cache maintenance. This work has yet to be published, although a prose version of the mathematics appears in the latest manuals [68]. It is difficult to give a comprehensive comparison between the most up-to-date model produced by Arm and the one presented here without further work to do an in-depth comparison. We do not believe the architectural intent has changed.

Operational instruction fetching

4.1 An Operational Semantics for Instruction Fetch

Previous work on operational models for IBM Power and Arm ‘user-mode’ concurrency (see Chapter 2) has shown, perhaps surprisingly, that one can capture the architecturally intended envelope of programmer-visible behaviour while abstracting from almost all hardware implementation details of the memory system (store queues, the cache hierarchy, the cache protocol, and so on). For Arm-A, following their 2018 shift to a multi-copy-atomic architecture, one can do so completely: the Flat model of Pulte, Flur, et al. [7] has a shared flat memory, with a per-thread out-of-order thread subsystem. This out-of-order thread subsystem abstractly models pipeline effects, which are alone sufficient to explain all the observable relaxed behaviours — subsuming relaxations which arise from store queues and caches and suchlike.

For instruction fetch, and the required cache maintenance, it is no longer possible to abstract completely from the data and instruction cache hierarchy. However, we can still abstract from some of its complexity. Flat has a fixed instruction memory, and fetches instructions from that fixed instruction memory. This fetch transition could be taken at any time, for any in-flight (non-finished) instruction, for any address of a potential (even speculative) program-order successor of that in-flight instruction. We now extend Flat by removing that fixed instruction memory, enabling instructions to be fetched from the flat memory, with values written by normal ‘data’ writes, along with adding the additional instruction-fetch related structures: per-thread fetch queues and instruction caches, and a global data cache, as shown in Figure 4.1.

We call this extended model *iFlat*. The remainder of this chapter will describe these new structures in detail, and enumerate the transitions of *iFlat*. We do so by first describing, informally, *iFlat* and its transitions, before giving a more detailed, but still in prose, precise description of the model. These descriptions are intended equivalent to a version we implement as an executable test oracle in the RMEM tool which can be found at <https://github.com/rem-project/rmem>.

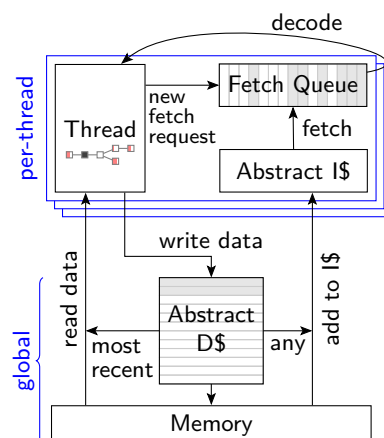


Figure 4.1: Structure of the *iFlat* state: per-thread fetch queues and instruction caches, with a global abstracted data cache.

4.2 iFlat Transitions: a brief summary

To Flat we add new transitions: splitting the atomic ‘fetch next instruction’ transition up into three distinct steps (a request, a satisfaction, and a decode); transitions which evolve the abstract cache state; and transitions for the operation of data and instruction cache maintenance.

In particular, these are¹:

- Fetch request
- ▷ Fetch instruction (ifetch)
- ▷ Fetch instruction (unpredictable)
- ▷ Fetch instruction (B.cond)
- Decode instruction
- ▷ Begin IC
- Propagate IC to thread
- ▷ Complete IC
- ▷ Perform DC
- Add to instruction cache for thread

In addition to these transitions, we modify some existing ones:

- ▷ Commit barrier
- ▷ Satisfy memory read by forwarding from writes
- ▷ Satisfy memory read from memory
- Commit store instruction
- ▷ Propagate memory write
- Complete store instruction (when its writes are all propagated)

Together, these transitions define the lifecycle of each instruction a request gets issued for the fetch, then at some later point the fetch gets satisfied from the instruction cache, the instruction is then decoded (in program-order), and then the ISA definition can be executed to produce intra-instruction traces to continue execution as in original Flat [7].

4.2.1 An example: DC/IC cache synchronisation

Before we describe the iFlat state and transitions in detail, we first give an informal intuition of how it works, with a walkthrough of the previously seen DC/IC sequence.

Figure 4.2 sketches the sequence of updates to the iFlat state over the cartoon sketch of Figure 4.1. It starts from a simple state, with an old cached value in the instruction cache and an already-satisfied element in the fetch queue. It then demonstrates the cache synchronisation sequence of: writing a new instruction, performing data cache maintenance, then instruction cache maintenance, before finally fetching the new instruction, in six steps:

1. A write of a new value is propagated to memory. The new write is placed into the abstract data cache buffer, where it can be seen by data reads but not guaranteed to be seen by instruction fetches.
2. The DC ‘pushes’ the write down, out of the abstract data cache, to memory.
3. The IC removes any cached copy of that location from the abstract instruction cache, for all cores. Also, it removes any already-satisfied elements in the fetch queue using those cached copies.
4. A spontaneous fill of the instruction cache is now guaranteed to see the new value, and will cache it in the abstract instruction cache.
5. Program-order later instructions for that location, which have not yet been fetched, can now be satisfied from the new entry in the instruction cache.
6. Once the satisfied fetch request reaches the end of the queue, it is decoded and appended to the end of the instruction tree in the thread to be executed.

Note that we do not include the thread-local effects of barriers (DMB, DSB, or ISB) in this sketch, see the full description of the transitions in §4.4 for further details.

¹Items marked with ○ may be taken eagerly.

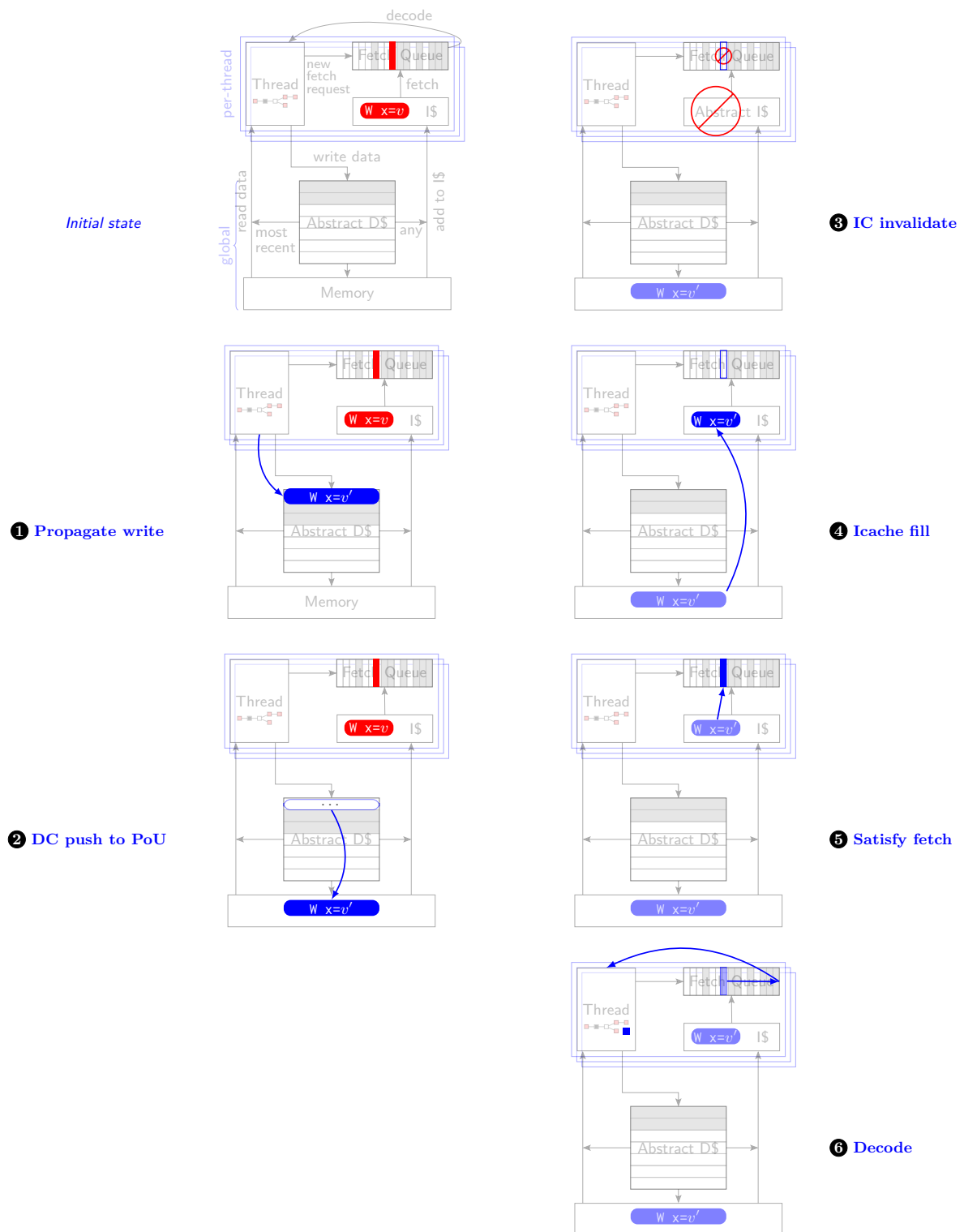


Figure 4.2: Cartoon sketch of DC/IC sequence in iFlat.

1853 4.3 The iFlat state

1854 We extend the original Flat state, to include:

- 1855 ▷ Per-thread fetch queues (§4.3.1), which store fetch requests permitting out-of-order satisfaction of
1856 instruction fetches.
- 1857 ▷ Per-thread instruction caches (§4.3.2), which enable threads to keep old (with respect to memory)
1858 copies of instructions which can be used when satisfying instructions in the fetch queue.
- 1859 ▷ A global abstracted data cache (§4.3.3), which over-approximates the coherent data cache network.

1860 As is usual for an architectural definition, these are all of unbounded size – abstracting from, and thus
1861 overapproximating, the hardware.

1862 The full description of the model state is given in [Appendix B](#).

1863 4.3.1 Fetch queues

1864 We give each thread a dedicated ‘fetch queue’, which buffers the in-flight instruction fetches. Fetch queues
1865 allow the model to speculate and pre-fetch instructions, potentially satisfying them out-of-order.

1866 The thread subsystem fetches instructions by inserting a new entry into the fetch queue. This entry is a
1867 *request*, containing the address to be fetched. The entries in the fetch queue can then be satisfied from
1868 memory at any point in time, in any order. Entries are removed and decoded in-order from the fetch
1869 queue.

1870 Entries are either a yet unsatisfied (‘*unfetched*’) request, or, a fetched 32-bit opcode.

1871 The model permits entries to be added to the fetch queue for any arbitrary address; as earlier instructions
1872 become finished, they will discard successor instructions whose program counter value does not match the
1873 one computed from the instruction semantics.

1874 In this way the fetch queues abstract from multiple hardware structures: instruction queues, line-fill
1875 buffers, loop buffers, slots objects, and others.

1876 **Out-of-order fetching** We believe the out-of-order satisfaction of instruction fetches is not observable on
1877 real hardware (in part due to the general lack of coherence in instruction caches subsuming this behaviour,
1878 see §3.5), and that the model is equivalent to one that fetches in order. However, this presentation of the
1879 model is more consistent with the description in the Arm reference manuals, and we believe has a closer
1880 correspondence with the underlying microarchitecture.

1881 **Fetch queues and instruction trees** Flat already keeps a per-thread tree of in-flight instructions. There
1882 is a model design choice between constructing an explicit fetch queue as an independent structure in the
1883 iFlat state, or adding a new *unfetched* state to the instruction instances in the tree and interpreting the
1884 po-suffix of any unfetched entries in the tree as the fetch queue. The latter has the advantage of allowing
1885 model speculation down multiple branches simultaneously, although this does not introduce additional
1886 behaviours.

1887 In principle, the two choices are equivalent and are merely presentational. Here, we give the model
1888 as if there are explicit separate fetch queues tracked in the state for exposition purposes, but in the
1889 executable-as-a-test-oracle implementation in RMEM we simply extend the already existing instruction
1890 tree with an unfetched state.

1891 4.3.2 Abstract instruction caches

1892 We give each thread an abstract instruction cache. The abstract instruction cache implements an over-
1893 approximation of those that may be found in hardware as a set of writes which the fetch queue entries
1894 can be satisfied from. We apply very few restrictions to the abstract instruction cache: it is permitted to
1895 cache any set of writes seen since the last instruction cache invalidation. The abstract instruction cache is
1896 then used to satisfy fetch requests in the fetch queue.

1897 The instruction cache can contain many possible writes for each location (§3.9), and can be spontaneously
1898 updated with new writes in the system at any time ([66, B2.4.4]), or spontaneously drop entries.

1899 Unlike the flat memory, the instruction caches are not touched on propagating a write. There is no guarantee
1900 values are ever dropped from the instruction cache, unless an explicit instruction cache maintenance
1901 operation is performed.

1902 **Invalidations** Instruction caches must be maintained by software, through the use of instruction cache
1903 invalidation instructions (IC). An IC instruction sends messages to each core (including its own), requesting
1904 they clear their instruction caches, and then waits for all the cores to reply. Other instructions may
1905 execute out-of-order with respect to these messages, except for DSBs: the requests are only sent after any
1906 program-order earlier DSB instructions are complete, and no program-order later DSB can complete until
1907 all the replies have returned.

1908 To handle this, we keep for each thread a set of addresses yet to be invalidated by in-flight ICs. Threads
1909 can, at any point in time, spontaneously perform any pending cache invalidation for that thread: removing
1910 any writes from the instruction cache or already-satisfied entries in the fetch queue, for that location.
1911 Once all pending cache invalidations have been performed on all cores, the IC instruction can complete.

1912 4.3.3 Global abstract data cache

1913 Above the single shared flat memory for the entire system, we insert a shared buffer of writes. Implemen-
1914 tations may have various topologies of cache hierarchies; we abstract from this by keeping a single large
1915 global buffer of all writes which have not yet passed the Point of Unification – over-approximating any
1916 individual cache topology.

1917 Explicit reads (e.g. those from load instructions) must be coherent, reading from the most recent write
1918 to the same address in the buffer, or memory. Instruction fetches may read from any write of the same
1919 location, from either the buffer or memory (§3.4).

1920 As writes are propagated to memory, they are placed initially into the abstract data cache buffer. At any
1921 point in time, the coherence-earliest write in the buffer for any location can spontaneously flow into the
1922 shared flat memory, making coherence-earlier writes no longer visible to instruction fetches.

1923 In this way, the shared flat memory acts as a system-wide Point of Unification; writes before that point
1924 may or may not be seen by the threads, but once they reach the shared flat memory an instruction cache
1925 fill must see that write, or something coherence newer.

1926 4.3.4 Outcome types

1927 To link the model transitions to the execution of the instructions in the program, the interface’s outcome
1928 types (described in §2.2) must be extended to cope with the new instructions: namely, we must add
1929 outcomes for the two cache maintenance operations, one for the data cache clean, and two for instruction
1930 cache invalidation (for the separation of propagation of messages and completion of the whole invalidation).
1931 The full list of outcomes for the iFlat model can be found in Figure 4.3.

READ_MEM(read_kind, address, size, read_continuation)	Read request
PERFORM_IC(address, res_continuation)	Propagate an ic ivau
WAIT_IC(address, res_continuation)	Wait for an ic ivau to complete
PERFORM_DC(address, res_continuation)	Propagate a dc cvau
WRITE_EA(write_kind, address, size, next_state)	Write effective address
WRITE_MEMV(memory_value, write_continuation)	Write value
BARRIER(barrier_kind, next_state)	Barrier
READ_REG(reg_name, read_continuation)	Register read request
WRITE_REG(reg_name, register_value, next_state)	Write register
INTERNAL(next_state)	Pseudocode internal step
DONE	End of pseudocode

Figure 4.3: iFlat outcomes (new outcomes highlighted in blue).

1932 4.3.5 Pseudocode states

1933 We extend the intra-instruction semantics, and associated pseudocode states, to include the fetch-queue
1934 fetch states, either fetched or unfetched, and ‘pending’ IC instructions, as they do not happen atomically.
1935 Figure 4.4 lists all the pseudocode states in iFlat, with the new ones highlighted.

PLAIN(next_state)	Ready to make a pseudocode step
UNFETCHED(pc)	Placed into fetch queue but pending satisfaction of the fetch
FETCHED(opcode)	Fetch satisfied but not yet begun pseudocode execution
PENDING_MEM_READS(read_cont)	Performing the read(s) from memory of a load
PENDING_MEM_WRITES(write_cont)	Performing the write(s) to memory of a store
PENDING_IC(ic_cont)	Waiting for completion of an IC IVAU

Figure 4.4: iFlat pseudocode states (new states highlighted in blue).

1936 4.4 iFlat Transitions: in full

1937 We now give full descriptions of the new transitions, as an amendment to the Flat model of [7] recalled in
1938 Chapter 2. A full transcription of the (i)Flat model can be found in Appendix B.

1939 4.4.1 New transitions

1940 Transitions for all instructions:

- 1941 ○ **Fetch request**: This transition (perhaps speculatively) requests to fetch the next-instruction address,
1942 as a po-successor of a previous instruction.
- 1943 ▷ **Fetch instruction**: Satisfy the fetch request from the instruction cache (split into three variants).
- 1944 ○ **Decode instruction**: Decode the instruction.

1945 Cache maintenance instructions:

- 1946 ▷ **Begin IC**: Initiate instruction cache maintenance.
- 1947 ○ **Propagate IC to thread**: Do instruction cache maintenance for a specific thread.
- 1948 ○ **Complete IC**: Finish instruction cache maintenance instruction.
- 1949 ▷ **Perform DC**: Clean the abstract data cache for a specific cache line.

1950 Instruction cache updates:

- 1951 ▷ **Add to instruction cache for thread**: Update instruction cache for thread with write.

1952 Below gives precise prose descriptions of each of the new transitions’ guard and action.

1953 **Fetch request** For some instruction i , any possible next fetch address loc can be requested, adding it to
1954 the fetch queue, if:

- 1955 1. it has not already been requested, i.e., none of the immediate successors of i in the thread’s
1956 instruction_tree are from loc ; and
- 1957 2. either i is not decoded, or, if it has been, loc is a possible next fetch address for i :
 - 1958 (a) for a non-branch/jump instruction, the successor instruction address ($i.program_loc+4$);
 - 1959 (b) for a conditional branch, either the successor address or the branch target address¹; or
 - 1960 (c) for a jump to an address which is not yet determined, any address (this is approximated in our
1961 tool implementation, necessarily).

1962 Action: add an unfetched entry for loc to the fetch queue for i ’s thread.

1964 Note that this allows speculation past conditional branches and calculated jumps.

¹In AArch64, all the conditional branch instructions have statically determined addresses.

1965 **Fetch instruction (ifetch)** *In ifetch mode this transition replaces the original ‘Fetch instruction’ transi-*
1966 *tion.*

1967 For any fetch-queue entry in the UNFETCHED state, its fetch can be satisfied from the instruction cache,
1968 from write-slices ws, if:

1969 1. the write-slices (parts of writes) ws have the 4-byte footprint of the entry and can be constructed
1970 from a write in the instruction cache.

1971 Action: change the fetch-queue entry’s state to FETCHED(ws).

1973 **Fetch instruction (unpredictable)** For any fetch-queue entry in the UNFETCHED state, its fetch can be
1974 satisfied from the instruction cache in a constrained-unpredictable way, if:

1975 1. there exists a set of write-slices, each of which can be constructed in the same way as above;
1976 2. that set contains write-slices corresponding to distinct opcodes, and at least one of those is an
1977 instruction that is not B.cond or one of {B, BL, BRK, HVC, SMC, SVC, ISB, NOP}, and they are not all
1978 B.cond instructions.
1979

1980 Action: record the fetch-queue entry as CONSTRAINED_UNPREDICTABLE. When this has reached decode
1981 and the corresponding point in the instruction tree becomes non-speculative, the entire thread state will
1982 become CONSTRAINED_UNPREDICTABLE.

1983 **Fetch instruction (B.cond)** For any fetch-queue entry in the UNFETCHED state, its fetch can be satisfied
1984 from the instruction cache, from write-slices ws and ws', with value ws'', if:

1985 1. there exists write-slices ws and ws', each of which can be constructed in the same way as above;
1986 2. ws and ws' correspond to the encoding of two conditional branch instructions b and b';
1987 3. the write-slices ws'' can be constructed as the combination of ws and ws' such that ws'' is the
1988 encoding of the branch instruction with b's condition and b's target.
1989

1990 Action: record the fetch-queue entry as FETCHED(ws'').

1991 **Decode instruction** If the last entry in the fetch queue is in FETCHED(ws) state, it can be removed from
1992 the queue, decoded, and begin execution, if all po-previous ISB instructions in the instruction tree have
1993 finished.

1994 Action:

1995 1. Construct a new instruction instance i with the correct instruction kind and state, for i's program
1996 location, and add it to the instruction tree.
1997 2. Discard all speculative entries in the instruction tree that are successors of i that are now known to
1998 be incorrect speculations.

1999 Note that this transition is a proxy for the point the instructions will be decoded, but that it is the
2000 intra-instruction semantics that actually performs the decoding, with this transition merely starting the
2001 execution of the pseudocode.

2002 **Begin IC** An instruction i (with unique instruction instance ID *iiid*) in state PERFORM_IC(address,
2003 state_cont) can begin performing the IC behaviour if all po-previous DSB ISH instructions have finished.
2004 Action:

2005 1. For each thread *tid'* (including this one), add (*iiid*, address) to that thread's ic_writes;
2006 2. Set the state of i to PROPAGATE_IC(address, state_cont).
2007

2008 **Propagate IC to thread** An instruction i (with ID *iiid*) in state WAIT_IC(address, state_cont) can
2009 do the relevant invalidate for any thread *tid'*, modifying that thread's instruction cache and fetch queue,
2010 if there exists a pending entry (*iiid*, address) in that thread's ic_writes.

2011 Action:

2012 1. For any entry in the fetch queue for thread *tid*, whose program_loc is in the same minimum-size
2013 instruction cache line as address, and is in FETCHED(____) state, set it to the UNFETCHED state.
2014 2. For the instruction cache of thread *tid*, remove any write-slices which are in the same instruction
2015 cache line of minimum size as address.
2016

2017 **Complete IC** An instruction i (with instruction instance ID $iiid$) in the state $WAIT_IC(address, state_cont)$ can complete if there exists no entry for $iiid$ in any thread's ic_writes .

2018 Action: set the state of i to $PLAIN(state_cont)$.

2020 **Perform DC** An instruction i in the state $PERFORM_DC(address, state_cont)$ can complete if all po-previous DMB ISH and DSB ISH instructions have finished.

2021 Action:

- 2023 1. For the most recent write slices wss which are in the same data cache line of minimum size in the
2024 abstract data cache as $address$, update the memory with wss .
- 2025 2. Remove all those writes from the abstract data cache.
- 2026 3. Set the state of i to $PLAIN(state_cont)$.

2028 **Add to instruction cache for thread** A thread tid 's instruction cache can be spontaneously updated with a write w from the storage subsystem, if this write (as a single slice) does not already exist in the instruction cache.

2029 Action: Add this write (as a single slice) to thread tid 's instruction cache.

2032 4.4.2 Updated transitions

2033 For those transitions where we update the guard or action the full text of the transition is reproduced here, with the delta highlighted, even where the change is minor.

2035 **Commit barrier** A barrier instruction i in state $PLAIN(next_state)$ where $next_state$ is $BARRIER(barrier_kind, next_state')$ can be committed if:

- 2037 1. all po-previous conditional branch instructions are finished;
- 2038 2. all po-previous `dmb sy` barriers are finished;
- 2039 3. `[ifetch]` all po-previous `dsb sy` barriers are finished; and
- 2040 4. if i is an `isb` instruction, all po-previous memory access instructions have fully determined memory footprints; and
- 2041 5. if i is a `dmb sy` instruction, all po-previous memory access instructions and barriers are finished;;
- 2042 and
- 2043 6. `[ifetch]` if i is a `dsb sy` instruction, all po-previous memory access instructions, barriers, and cache maintenance instructions have finished.

2044 Action:

- 2047 1. Update the state of i to $PLAIN(next_state')$;
- 2048 2. `[ifetch]` If i is an `isb` instruction, for any instruction instance in this thread's instruction tree, if that instruction instance is in the `FETCHED` state, set it to the `UNFETCHED` state.

2049 Note that this corresponds to an ISB discarding any already-fetched entries from the fetch queue.

2053 **Satisfy memory read by forwarding from writes** For a load instruction instance i in state $PENDING_MEM_READS(k)$, and a read request, r in $i.mem_reads$ that has unsatisfied slices, the read request can be partially or entirely satisfied by forwarding from unpropagated writes by store instruction instances that are po-before i , if the *read-request-condition* predicate holds. This is if:

- 2057 1. `[ifetch]` all po-previous `dsb sy` instructions are finished; and
- 2058 2. all po-previous `dmb sy` and `isb` instructions are finished.

2059 Let wss be the maximal set of unpropagated write slices from store instruction instances po-before i , that overlap with the unsatisfied slices of r , and which are not superseded by intervening stores that are either propagated or read from by this thread. That last condition requires, for each write slice ws in wss from instruction i' :

- 2060 \triangleright that there is no store instruction po-between i and i' with a write overlapping ws , and
- 2061 \triangleright that there is no load instruction po-between i and i' that was satisfied from an overlapping write slice from a different thread.

2067 Action:

2068

2069

2070

2071

2072

2073

2074

2075

1. Update r to indicate that it was satisfied by wss .
2. Restart any speculative instructions which have violated coherence as a result of this, i.e., for every non-finished instruction i' that is a po-successor of i , and every read request r' of i' that was satisfied from wss' , if there exists a write slice ws' in wss' , and an overlapping write slice from a different write in wss , and ws' is not from an instruction that is a po-successor of i , or if i' was a data-cache maintenance by virtual address to a cache line that overlaps with any of the write slices in wss' , restart i' and its data-flow dependents.

2076

2077

2078

Satisfy memory read from memory For a load instruction instance i in state `PENDING_MEM_READS(k)`, and a read request r in $i.mem_reads$, that has unsatisfied slices, the read request can be satisfied from memory, if:

2079

1. the read-request-condition holds (see previous transition).

2080 Action:

2081

2082

2083

let wss be the write slices from memory or the abstract data cache, whichever is newer, covering the unsatisfied slices of r , and apply the action of **Satisfy memory read by forwarding from writes**.

2084

2085

2086

Note that **Satisfy memory read by forwarding from writes** might leave some slices of the read request unsatisfied. **Satisfy memory read from memory**, on the other hand, will always satisfy all the unsatisfied slices of the read request.

2087

2088

Commit store instruction For an uncommitted store instruction i in state `PENDING_MEM_WRITES(k)`, i can commit if:

2089

2090

2091

2092

2093

2094

2095

2096

1. i has fully determined data (i.e., the register reads cannot change);
2. all po-previous conditional branch instructions are finished;
3. all po-previous `dmb sy` and `isb` instructions are finished;
4. `[ifetch]` all po-previous `dsb sy` instructions are finished;
5. all po-previous store instructions have initiated and so have non-empty `mem_writes`;
6. all po-previous memory access instructions have a fully determined memory footprint; and
7. all po-previous load instructions have initiated and so have non-empty `mem_reads`.

2097

Action: record i as committed.

2098

2099

Propagate memory write For an instruction i in state `PENDING_MEM_WRITES(k)`, and an unpropagated write, w in $i.mem_writes$, the write can be propagated if:

2100

2101

2102

2103

2104

1. all memory writes of po-previous store instructions that overlap w have already propagated;
2. all read requests of po-previous load instructions that overlap with w have already been satisfied, and the load instruction is non-restartable; and
3. all read requests satisfied by forwarding w are entirely satisfied.

2105

Action:

2106

2107

2108

2109

2110

2111

2112

2113

1. Restart any speculative instructions which have violated coherence as a result of this, i.e., for every non-finished instruction i' po-after i and every read request r' of i' that was satisfied from wss' , if there exists a write slice ws' in wss' that overlaps with w and is not from w , and ws' is not from a po-successor of i , or if i' is a data-cache maintenance instruction to a cache line whose footprint overlaps with w , restart i' and its data-flow dependents.
2. Record w as propagated.
3. Add w as a complete slice to the abstract data cache.

2114

4.4.3 Auxiliary definition – cache line of minimum size

2115

2116

2117

2118

Cache maintenance operations work over entire cache lines, not individual addresses (§3.12). Each address is associated with at least one cache line for the data (and unified) caches, and one for the instruction caches. The data and instruction cache line of minimum size is the smallest possible cache line, for the data or instruction caches respectively. The `CTR_EL0.{DMinLine, IMinLine}` register fields describe the

2119 cache lines of minimum size for the data and instruction caches as \log_2 of the number of words in the
2120 cache line.

2121 Caches lines are always aligned on their minimum size, and we say a write slice *overlaps* with a cache line
2122 if the footprint of the write slice overlaps with the $2^{2+DMinLine}$ (or $2^{2+IMinLine}$ for instruction cache lines)
2123 byte slice starting from the beginning of the aligned cache line region.

2124 **4.4.4 Handling cache type strengthenings**

2125 When `CTR_EL0.DIC` is 1, and therefore the IC instruction is not required, the following transitions are
2126 modified:

- 2127 ▷ **Fetch instruction:**
 - 2128 – Instead of satisfying from the instruction cache, the request must be satisfied from composing
 - 2129 combinations of writes from the abstract data cache buffer and flat memory.
 - 2130 – Fetch requests may be only be satisfied if all po-previous in-flight fetch requests are also satisfied
 - 2131 (no out-of-order satisfaction).
- 2132 ▷ **Fetch instruction (unpredictable)** (same modification as previous).
- 2133 ▷ **Fetch instruction (B.cond)** (same modification as previous).
- 2134 ▷ **Begin IC:**
 - 2135 – Replace action with that of **Complete IC**.
- 2136 ▷ **Add to instruction cache for thread** (removed).

2137 Together these effectively remove the instruction cache from the model, forcing in-order fetching, and
2138 satisfaction of fetch requests from memory (or the abstract data cache).

2139 When `CTR_EL0.IDC` is 1, and therefore the DC instruction is not required, the following transitions are
2140 modified:

- 2141 ▷ **Propagate memory write:**
 - 2142 – Update Action (3) to add w to the flat memory, instead of the abstract data cache buffer.

2143 This effectively removes the abstract data cache buffer from the model, causing all writes to immediately
2144 reach the system-wide Point of Unification on propagation.

An axiomatic instruction fetch model

Based on the operational model, we develop an axiomatic semantics, as an extension of the Arm-A axiomatic model [53, 7] described in Chapter 2. Throughout this chapter, references to the base Arm-A axiomatic model refer to the one presented in that chapter.

The base axiomatic model is given as a predicate on *candidate executions*, hypothetical complete executions of the given program which satisfy some basic well-formedness conditions, defining the set of *valid* executions to be those satisfying its axioms.

We now extend this model, extending both the base events and candidate relations, as well as modifying the axioms over those events. We do this in a way that tries to retain the original model events, relations, and axioms, as unchanged as is reasonable to do so.

5.1 Candidates for self-modifying programs

We add new events:

- ▷ *instruction-fetch* (IF) events for each executed instruction, corresponding to the read of the 32-bit opcode from memory during an instruction fetch.
- ▷ DC events, corresponding to the propagation of cache maintenance from a DC CVAU instruction.
- ▷ IC events, corresponding to the propagation of cache maintenance from an IC IVAU or IC IALLU instruction.
- ▷ DSB events for the data synchronization barrier instruction.

5.1.1 Explicit events and program order

We partition the events into *implicit* events (in this case, just instruction fetches) and *explicit* events. Logically, the explicit events are those that form part of the intensional operation of the instruction: the primary memory events and general-purpose register accesses; whereas the implicit events are those accesses which are indirect, part of the execution model of the machine or configuration but not particular to the behaviour of that instruction. The architecture does not precisely define which events of an execution belong to which category, but the informal notion of an ‘implicit’ event will still be a useful one.

Program order (po) relates explicit events. These are the explicit memory accesses (non-instruction-fetch), barriers (including DSBs), and the cache operations (DC, IC).

By adding an instruction fetch event we now potentially have multiple events per instruction. To keep track of the order of events within a single instruction, and between multiple instructions of the same thread, we add two new relations:

- ▷ *fetch-to-execute* (fe), which relates the instruction fetch (IF) event with the intra-instruction-ordered-later explicit memory access, barrier, or cache-op events of the instruction.
- ▷ *fetch-program-order* (fpo), which relates each instruction-fetch (IF) event with all IF events of program-order later instructions.

2180 We make fpo the fundamental relation in candidates, instead of po , which we instead derive:

$$po = fe^{-1}; fpo+; fe$$

2181 Figure 5.1 shows an example execution graph from a program with three instructions a load, a move, and,
2182 a store; with the implicit fetch events highlighted, illustrating the derivation of po .

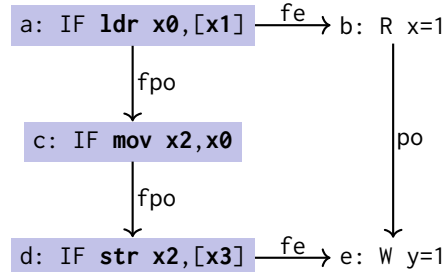


Figure 5.1: An example showing how po is derived from fpo and fe from the implicit fetch events (marked in blue).

2183 5.1.2 Same-location

2184 We extend loc to relate same-address reads, writes, instruction fetches and IC/DC events.

2185 Cache maintenance operations which affect all addresses, for example the IC IALLU instruction, are related
2186 to all memory and ifetch events.

2187 **Same-cache-line** Many of the operations now operate not over a single location but an entire cache line.
2188 To handle these operations, we add to the candidate relations a pair of *same-cache-line* relations, relating
2189 reads, writes, fetches, DC, and IC events to addresses in the same cache line of minimum size.

2190 Since the DC and IC instructions operate over different cache line sizes, we have separate *same-dcache-line*
2191 and *same-icache-line* relations, to relate events in the same data or instruction cache line of minimum
2192 size. Note that the *same-icache-line* and *same-dcache-line* relations also relate non-cache-op events.

2193 We combine these relations to get a single scl (same cache line), between memory (including ifetch) events
2194 and cache ops, where that memory event is to the same cache line, for that particular cache op:

```

2195 1 scl0 = [DC]; same-dcache-line | [IC]; same-icache-line | [W]; loc
2196 2 scl = scl0 | scl0-1

```

2197 5.1.3 Generalised Coherence

2198 We add an acyclic, transitively closed, relation, wco . This is a generalised coherence-order, an extension of
2199 co , with orderings for cache maintenance (DC and IC) events: it includes an ordering (e, e') or (e', e) for
2200 any cache maintenance event e and same-cache-line event e' if e' is a write or another cache maintenance
2201 event.

2202 Since wco relates events in the same cache line, and is transitively closed, it may end up relating writes
2203 that are not the same location. So $[a:W]; wco; [b:W]$ does not imply $[a:W]; co; [b:W]$ (although co does
2204 imply wco).

2205 This relation forms part of the witness, and abstractly captures the order that cache maintenance operations
2206 and propagation of writes would happen in the operational model.

2207 5.1.4 Dependencies

2208 We extend the control dependency relation $ctrl$ to include cache operations, but not instruction fetches.
2209 This ensures that $ctrl$ remains a subset of po , and that $[a]; ctrl; [b]; po; [c]$ implies $[a]; ctrl; [c]$.

2210 We extend `addr` to include cache operations, so that $(e, e') \in \text{addr}$ when e is a read and e' is a cache
 2211 operation (DC or IC) whose address (cache line) is determined by the value read by e .

2212 Since cache operations do not have any data associated with them, the `data` relation is left unchanged.

2213 5.1.5 Reads-from

2214 We add an *instruction-read-from* (`irf`) relation to the witness. It is the analogue of `rf` for instruction
 2215 fetches, relating writes to the IF event that fetches from it. We derive the analogous from-reads relation,
 2216 *instruction-from-reads* (`ifr`), from a fetch to all writes coherence-after the one it fetched from:

$$\text{ifr} = \text{irf}^{-1}; \text{co}$$

2217 Note the use of `co` not `wco`.

2218 5.2 Axioms and auxiliary relations

2219 We now make the following changes and additions to the model. The full model is shown in Figure 5.2,
 2220 with comments referring to the items in the following explanation.

2221 5.2.1 Arm ifetch events and relations

2222 The `arm-common.cat` file contains all the Arm-specific event names and relations, as defined in Chapter 2,
 2223 and can be found in the full isla sources for these models in [91]. Figure 5.3 lists the events and relations
 2224 defined by that file; we elide the full isla-cat definition of these relations here.

Events	Relations
R Reads	<code>po, fpo</code> program-order and <code>fetch-program-order</code>
IF <i>Instruction-fetch</i>	<code>id, loc</code> identity and same-location
W Writes	<code>fe</code> <i>fetch-to-execute</i>
M Explicit memory event (R W)	<code>po-loc</code> program-order same-location (<code>po & loc</code>)
A Read-acquire	<code>addr, ctrl, data</code> dependencies
L Write-release	<code>wco, irf, rf</code> Witness relations
Q Weak read-acquire	<code>rfe, rfi</code> rf-external (<code>rf&ext</code>), rf-internal (<code>rf&~ext</code>)
F All fences (barriers)	<code>coe, coi</code> co-external, co-internal
C All cache-ops (DC IC)	<code>co</code> coherence-order (<code>[W]; wco&loc; [W]</code>)
DC Data cache clean	<code>ifr</code> <i>instruction-from-reads</i> (<code>irf^{-1}; co</code>)
IC Instruction cache invalidate	<code>rmw</code> read-modify-write
ISB Instruction barrier	
dmbXY Memory Barrier	
dsbXY DSB Barrier	
	<code>scl</code> same-cache-line
	<code>same-dcache-line, same-icache-line</code> same data/instruction cache line

Variants

DIC, IDC Boolean flags for `CTR_EL0.{DIC, IDC}` identity

Figure 5.3: Arm ifetch events and relations. New and updated are highlighted in blue.

2225 5.2.2 Cache maintenance

2226 We define a derived relation `isyncob` (*instruction-synchronised-ordered-before*), relating some instruction
 2227 fetch f , in the most general case, to an IC which completes a cache synchronisation sequence (not
 2228 necessarily on a single thread) which affects the location fetched. Consequently, any instruction fetch

```

1  include "cos.cat"
2  include "arm-common.cat" (*5.2.1*)
3
4  (* might-be speculatively executed *)
5  let speculative =
6      ctrl
7      | addr; po
8
9  (* Fetch-ordered-before *)
10 let fob =
11     [IF]; fpo; [IF] (*5.2.4*)
12     | [IF]; fe (*5.2.4*)
13     | [ISB]; fe-1; fpo (*5.2.5*)
14
15 (* Cache-op-ordered-before *)
16 let cob = (*5.2.8*)
17     [R|W]; (po & scl); [DC]
18     | [DC]; (po & scl); [DC]
19
20 (* DC synchronised required after a write *)
21 let dcsync =
22     if IDC
23     then id
24     else [W]; (wco & same-dcache-line); [DC]
25
26 (* IC sync required after a write or DC *)
27 let icsync =
28     if DIC
29     then id
30     else (
31         [W]; (wco & same-icache-line); [IC]
32         | [DC]; wco; [IC]
33     )
34
35 let cachesync =
36     dcsync; icsync
37
38 (* instruction-synchronised-ordered-before *)
39 let isyncob = (*5.2.2*)
40     (ifr; cachesync) & scl-1
41
42     (* observed by *)
43     let obs = rfe | fr | wco | irf
44
45     (* dependency-ordered-before *)
46     let dob =
47         addr | data
48         | speculative; [W]
49         | speculative; [ISB]
50         | (addr | data); rfi
51
52     (* atomic-ordered-before *)
53     let aob =
54         rmw
55         | [range(rmw)]; rfi; [A|Q]
56
57     (* barrier-ordered-before *)
58     let bob =
59         [R]; po; [dmbld]
60         | [W]; po; [dmbst]
61         | [dmbst]; po; [W]
62         | [dmbld]; po; [R|W]
63         | [L]; po; [A]
64         | [A|Q]; po; [R|W]
65         | [R|W]; po; [L]
66         | [F|C]; po; [dsbsy] (*5.2.6*)
67         | [dsb]; po (*5.2.6*)
68         | [dmsby]; po; [DC] (*5.2.7*)
69
70     (* Ordered-before *)
71     let ob1 =
72         obs | dob | aob | bob
73         | fob | cob | isyncob
74     let ob = ob1+
75
76     (* Internal visibility *)
77     acyclic po-loc | fr | co | rf
78     as internal
79
80     (* External visibility *)
81     irreflexive ob as external
82
83     (* Atomic *)
84     empty rmw & (fre; coe) as
85     atomic

```

Figure 5.2: Ifetch Axiomatic model

2229 must have been satisfied before the completion of any cache maintenance that it is `isyncob`-ordered before.
 2230 Precisely, f `isyncob` i is defined as:

$$f \text{ isyncob } i \iff (f, i) \in (\text{ifr}; \text{cachesync}) \cap \text{scl}$$

2231 That is, f reads-from some write w_0 , which was coherence-before some other write w , and w is `wco`-before
 2232 by a DC event d to some same-dcache-line address A_{dc} , which is in turn was `wco`-before by an IC event
 2233 i to some address A_{ic} which was same-icache-line as the original f . This general `isyncob` shape is
 2234 shown in Figure 5.4. In operational model terms, this captures traces that propagated w to memory, then
 2235 subsequently performed a same-cache-line DC, and then began an IC (and eagerly propagated the IC to all
 2236 threads). In any state after this sequence it is guaranteed that w , or a coherence-newer same-address
 2237 write, is in the instruction cache of all threads: performing the DC has cleared the abstract data cache
 2238 of writes to x , and the subsequent IC has removed old instructions for location x from the instruction
 2239 caches, so that any subsequent updates to the instruction caches have been with w , or `co`-newer writes.
 2240 Therefore, the fetch f must have happened before the IC had completed, otherwise it would have been
 2241 required to have read from w or something coherence after it.

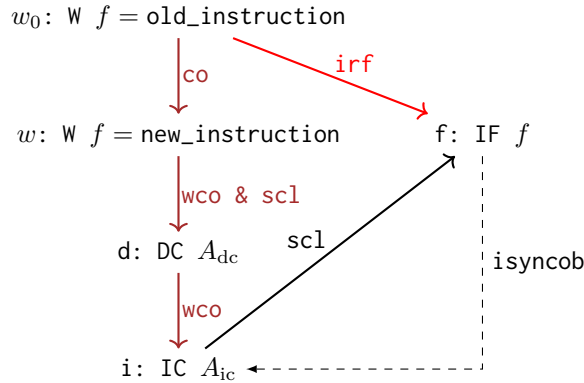


Figure 5.4: General `isyncob` shape.

2242 This corresponds to the operational model in the following way: because w_0 was coherence-before w , w_0
 2243 was propagated before w was propagated in the trace, and because w was `wco`-earlier than the cache
 2244 synchronisation sequence, w was propagated before any of the cache maintenance transitions in the trace.
 2245 If the fetch transition corresponding to f were to satisfy its fetch in a subsequent state, it would be
 2246 guaranteed that w (or a coherence-newer write) would be in the instruction cache, and i would not be
 2247 able to fetch from w . Hence, f must have happened before the IC completing the cache synchronisation
 2248 sequence.

2249 **Cache type strengthening** If the IDC or DIC variants are set, then either the DC or IC instruction is not
 2250 required. This affects the `isyncob` in the following way:

- 2251 ▷ If DIC, then the IC instruction is not required, and therefore f must be ordered before the propagation
 2252 of the DC, see Figure 5.5 (top left).
- 2253 ▷ If IDC, then the DC instruction is not required, and therefore f must be ordered before the propagation
 2254 of the IC, without the need of an intervening DC, see Figure 5.5 (top right).
- 2255 ▷ If both, then f must be ordered before any coherence-later same-location write than w_0 , as in
 2256 Figure 5.5 (below).

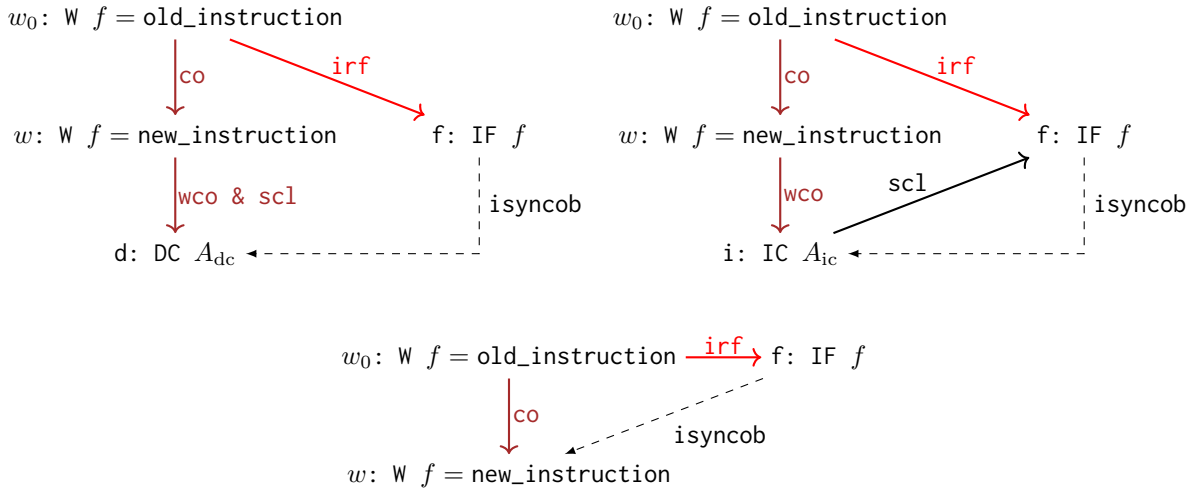


Figure 5.5: Modified `isyncob` shape, for variants DIC (above left), IDC (above right), and both (below).

2257 To achieve this, the `isyncob` relation is derived from the composition of two smaller relations:

- 2258 ▷ `dcsync`, which broadly captures the ‘data cache’ requirements, either from a write to a same cache
- 2259 line DC if not IDC, otherwise, from a write to itself, capturing that with IDC that a write is past the
- 2260 PoU the moment it has propagated.
- 2261 ▷ `icsync`, which captures the ‘instruction cache’ requirements, either from a DC (or `same-icache-line`
- 2262 write), to a `wco`-later IC, or, if DIC, back to the DC or write itself.

2263 The sequential composition of these two relations (called `cachesync`) captures the synchronisation required

2264 from a write to the point sufficient cache maintenance has been performed to ensure a same-cache-line

2265 instruction fetch would see it. We then finally define `isyncob` between any instruction fetch, and any

2266 cache maintenance operation which is `cachesync`-after any write coherence-after the one the fetch read

2267 from, that is, a write which has had sufficient cache synchronisation to have made earlier writes invisible

2268 to the fetch machinery.

2269 5.2.3 Coherence

2270 The original model includes `co` in `obs`; we instead include the relation `wco`. Including `wco` in ordered-before

2271 corresponds to the intuition that `wco` records the ordering of the `Propagate memory write`, `Begin IC` (and

2272 eagerly taking all `Propagate IC to thread` transitions), and `Perform DC` transitions in the matching trace.

2273 We also include `irf` in `obs`: informally, for an instruction to be fetched from a write, the write has to have

2274 been done before. Correspondingly, in the operational model, a write has to have been propagated before

2275 it can satisfy fetches in the storage subsystem.

2276 5.2.4 Program order

2277 We add a derived relation `fetch-ordered-before` (`fob`), which is included in `ordered-before`.

2278 The `fob` relation includes `fpo`, informally requiring fetches to be ordered according to their order in the

2279 control-flow unfolding of the execution. Correspondingly in the operational model: fetch requests for

2280 instructions within the same thread appear to be satisfied in program order.

2281 We also include the `fe` `fetch-to-execute` relation in `fob`, capturing the idea that an instruction must be

2282 fetched before it can execute. In the operational model, a read can only be satisfied, a write can only

2283 propagate, a barrier can only commit, and so on, after its instruction’s fetch is satisfied.

2284 5.2.5 Instruction synchronisation (ISB)

2285 We include the edge $[ISB]; fe^{-1}; fpo$ in fetch-ordered-before (*fob*), ordering the fetch of any instruction
2286 program-order-succeeding an ISB instruction after the ISB event.

2287 Operationally, a decoded ISB instruction prevents any program-order-later instructions from being removed
2288 from the fetch queue and decoded, and when an ISB is executed, it returns all entries in this thread's
2289 fetch queue (so any program-order-later instructions) to the UNFETCHED state, forcing the satisfaction of
2290 the instruction fetch for those instructions to happen after the ISB completes.

2291 The rule $[ISB]; po; [R]$ in *dob* is no longer required, as the combination of rules in *fob* (in particular
2292 $[ISB]; fe^{-1}; fpo$ and $[IF]; fe$) subsume it.

2293 5.2.6 Data synchronisation (DSB)

2294 For DSB instructions we include *po* to and from DSB in the standard barrier-ordered-before relation (*bob*).

2295 We do this in three ways: (1) by extending the barrier hierarchy relations *dmbst* and *dmbld* to cover
2296 the memory barrier effects of a DSB; (2) by adding $[F|C]; po; [dsbsy]$ to capture DSBs waiting for the
2297 completion of fences and cache-ops, when using DSBs affecting both reads and writes; and (3) by adding
2298 $[dsb]; po$ to capture the remaining completion fence properties that program-order later events cannot go
2299 ahead until the DSB is complete.

2300 Importantly, DSB events do not order IF (*ifetch*) events in either direction.

2301 5.2.7 Data cache maintenance (DC) is ordered like a read

2302 Barrier-ordered-before also includes the relation $[dmb sy]; po; [DC]$, ordering DC events after program-order-
2303 preceding DMB SYs. Correspondingly, in the operational model, a DC can only be performed when all
2304 preceding DMB SY are finished.

2305 5.2.8 Cache maintenance operations and cache lines

2306 We include the relation *cache-op-ordered-before* (*cob*) in *ob*. This relation contains the edge $[R|W]; (po \&
2307 sc1); [DC]$, ordering DC events after program-order-preceding same-dcache-line read and write events.

2308 Operationally, a DC will be restarted by a program-order-preceding same-cache-line load if it was performed
2309 before the load was satisfied, and by a program-order-preceding same-cache-line store if it was performed
2310 before the store propagated its write.

2311 Moreover, *cob* contains the edge $[DC]; (po \& sc1); [DC]$, ordering two same-cache-line, same-thread DC events
2312 in program-order. In the operational model, a DC can only be performed when program-order-preceding
2313 same-cache-line DC instructions have been performed.

2314 **5.2.9 Constrained Unpredictable**

2315 We do not give precise semantics to programs that exhibit constrained unpredictable behaviour. Instead,
 2316 we add a mechanism to flag such programs.

```

1  (* include base ifetch model *)
2  include "aarch64_ifetch.cat"
3
4  (* could-fetch-from *)
5  let cff =
6    ([W]; loc; [IF])
7    \ ob-1
8    \ (isyncob-1; ob)
9
10 (* cmodx(opcode) is True
11  * if it is in the list of
12   concurrently modifiable
13   instructions
14  *)
13 define cmodx(v: bits(32)): bool =
14   ...
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000

```

Figure 5.6: Constrained unpredictable check model (ifetch).

2317 We do this through the definition of an auxiliary *could-fetch-from* (*cff*) relation, capturing, for each fetch
 2318 *i*, the writes it could have fetched from (including the one it did fetch from), as the set of same-address
 2319 writes that are not ordered-after *i*, and which are not overwritten by coherence-newer writes that were
 2320 followed by a cachesync sequence ordered-before *i*. Operationally, this captures writes that could have
 2321 been in the instruction cache of *i*'s thread: writes that did not happen *after i* in the trace, and excluding
 2322 writes cleared by earlier cache synchronisation sequences.

2323 We then add an axiom, asserting the existence of a bad pair of writes (w_1, w_2) which *i* could have fetched
 2324 from, where at least one of w_1 and w_2 are not in the list of concurrently-modifiable instructions (as
 2325 described in §3.2). We identify these (i, w_1, w_2) triples with a ternary relation ($\text{cff_bad}(w_1, i, w_2)$), whose
 2326 non-emptiness implies the existence of such a triple. This gives us an extended ‘checker’ model, where
 2327 executions which are allowed in the checker model, are also allowed in the original ifetch model, but also
 2328 exhibit constrained unpredictable behaviour, and so the test should be flagged and any results discarded.
 2329

Validating the ifetch models

We gain confidence in the models presented in the previous chapters by validating those models against the Arm architectural intent, against each other, and against a selection of real hardware.

6.1 The models correctly captures the architectural intent

This property is an important one, but not one that can be objectively demonstrated.

We ensure that the models reflect the architecture, to the best of our understanding, by engaging in detailed and robust discussions with the Arm chief architect, as well as microarchitects involved in the design of individual processors.

This process is an iterative one, where we produce litmus tests, discuss whether they are allowed or forbidden (and by which mechanisms), build models that capture those described mechanisms, and produce more litmus tests that show edge cases or interactions. This process is not necessarily terminating, but it usually results in reaching a natural fixed point, for a core set of architectural features.

The structure of the operational model presented in [Chapter 4](#) is based on our discussions with Arm; it carefully includes structures which capture the behaviour they described, and has limits where the architects decided no reasonable hardware could exploit.

The axiomatic model, presented in [Chapter 5](#), is also a product of the discussions with Arm.

6.2 Correspondence between the models

We experimentally test the correspondence between the operational and axiomatic models. We do this by making the models executable as test oracles, and running a suite of hand-written and autogenerated litmus tests, checking that both models give the same result in all cases.

To automatically generate families of interesting instruction-fetch tests, Luc Maranget (a co-author of this work) extended the ‘diy’ test generation tool [70] to support instruction-fetch reads-from (irf) and instruction-fetch from-reads (ifr) edges, in both internal (same-thread) and external (inter-thread) forms, and the cachesync edge. We used this to generate 1456 tests involving those edges together with po, rf, fr, addr, (but no data), ctrl, ctrlisb, and dmb.sy. diy does not currently support bare DC or IC instructions, locations which are both fetched and read from, or repeated fetches from the same location.

6.2.1 Making the operational model executable as a test oracle

To make the operational model presented in [Chapter 4](#) executable, that is, capable of computing the set of all allowed executions of a litmus test, we must be able to *exhaustively enumerate* all possible traces. For the model as presented, doing this naively is infeasible: for each instruction it is theoretically possible to speculate any of the 2^{64} addresses as the address of a potential successor instruction, and the interleaving of the new fetch transitions with others leads to an additional combinatorial explosion.

2363 We address these with two new optimisations. First, we extend the fixed-point optimisation in RMEM,
 2364 which incrementally builds the set of possible branch targets by repeated exhaustive searches [7], to track
 2365 not only the indirect branch instructions but the successors of *every* program location. Additionally, we
 2366 track during a test which locations were both fetched and modified during the test, and eagerly take fetch
 2367 and decode transitions for all other locations. As before, the search then runs until the set of branch
 2368 targets *and* the set of modified program-locations reaches a fixed point.

2369 Confluence

2370 We also take some transitions eagerly when they cannot remove behaviour to reduce the search space:
 2371 ‘Propagate IC to thread’, ‘Complete IC’, ‘Fetch request’, and ‘Add to instruction cache for thread’.

2372 Taking ‘Add to instruction cache for thread’ eagerly is ok, as this always increases the visible behaviours:
 2373 adding a write to an instruction cache does not hide writes that were visible before. ‘Complete IC’ and
 2374 ‘Fetch request’ are also safe to take eagerly, as these advance thread-local state in a way that makes
 2375 further transitions available without preventing any others.

2376 Taking ‘Propagate IC to thread’ eagerly is more subtle; this transition updates the state of another thread
 2377 and potentially removes transitions it had available to it. If we take an arbitrary trace, containing a
 2378 propagation of an IC to some thread, then it is safe (by the aforementioned logic) to immediately fill
 2379 that cache back in. If we have a trace with two IC propagations, to separate threads, from the same
 2380 instruction, with propagations of writes and DCs in between, then we know that the second ‘Propagate IC
 2381 to thread’ must have been an available transition when taking those write and DC propagation transitions,
 2382 and therefore there must have been another trace where those write and DC propagations happened *after*
 2383 the second IC propagation, and where the icache is filled immediately after each of those writes.

```
2384 ...
2385   Propagate IC to X on Thread 1
2386   Write to X
2387   Propagate DC to X
2388   Write to X
2389   Propagate IC to X on Thread 2
2390   ...
```

2391 ⇒

```
2392 ...
2393   Propagate IC to X on Thread 1
2394   Propagate IC to X on Thread 2
2395   Write to X
2396   Eagerly fill icache
2397   Propagate DC to X
2398   Write to X
2399   Eagerly fill icache
2400   ...
```

2401 This new trace groups the propagation of instruction cache invalidations together as early as possible,
 2402 maximising the visible behaviour. Therefore, it we perform all the invalidates at once, atomically.

2403 6.2.2 Making the axiomatic model executable as a test oracle

2404 We give the axiomatic model in the `isla-cat` memory modelling language (see §2.4.2).

2405 As `isla-axiomatic` already executes a fetch-decode-execute loop, defined by the Arm intra-instruction
 2406 semantics, the changes required of the ISA definition are only minor; we need only create outcomes for
 2407 the fetch memory accesses, and pass them as events to the axiomatic model.

2408 This is sufficient for making the test executable, but exhaustive enumeration becomes intractable, as
 2409 the fetch events in the candidates should, in theory, be totally unconstrained. To support exhaustive
 2410 enumeration we must reduce the set of candidates we are required to check. Even permitting the *fetch*
 2411 part of the loop to be entirely symbolic (in location and opcode) would lead to far too many candidate
 2412 executions. Even if the vast majority of them would be dismissed quickly, with trivially unsatisfiable `irf`

2413 constraints they would still take time to generate and discharge. To avoid this, we instead require the user
 2414 to provide the possible set of program-counter values, and the sets of opcodes those locations' values can
 2415 be. This ensures that while generating candidates we only need to generate those that actually contain
 2416 the control-flow and instruction opcodes that are interesting for the test.

2417 Figure 6.1 contains the `isla-axiomatic-compatible` sources for the earlier `SM.F+ic` test (Figure 3.19, p.55)
 2418 as an example. Lines 7-13 define the self-modifiable locations used in the test (for this test that is only
 2419 label 'f:'), and the fully-concrete opcodes those locations may be; recall that all `isla` traces are a single
 2420 control-flow path with fully concrete opcodes for each instruction.

```

1  arch = "AArch64"
2  name = "SM.F+ic"
3  hash = "de102a920be43ce10482e59700a7c976"
4  stable = "X10"
5  symbolic = ["x"]
6
7  [[self_modify]]
8  address = "f:"
9  bytes = 4
10 values = [
11   "0x14000001",
12   "0x14000003"
13 ]
14
15 [thread.0]
16 init = { X3 = "x", X4 = "f:", X0 = "0x14000001" }
17 code = ""
18     STR W0,[X4]
19     LDR W2,[X3]
20     CBZ W2, 1
21 1:
22     ISB
23     BL f
24     MOV W1,W10
25     B Lout
26 f:
27     B 10
28 11:
29     MOV W10,#2
30     RET
31 10:
32     MOV W10,#1
33     RET
34 Lout:
35 ""
36
37 [thread.1]
38 init = { X3 = "x", X2 = "1", X1 = "f:" }
39 code = ""
40     BLR X1
41     MOV W0,W10
42     IC IVAU, X1
43     DSB SY
44     STR W2,[X3]
45 ""
46
47 [final]
48 expect = "sat"
49 assertion = "1:X0 = 2 & 0:X2 = 1 & 0:X1 = 1"

```

Figure 6.1: Test `SM.F+ic` `isla-axiomatic compatible` version.

2421 6.3 Equivalence of the models

2422 Ideally, one would have a formal proof that the operational and axiomatic models coincide, or at least a
2423 detailed proof of some properties we expect the operational model to have: that the model is equivalent
2424 to one that fetches in-order, that the transitions we take eagerly are safe to do so, that the fixed-point
2425 calculation is not unsound for the model, and so on. However, this represents a large undertaking, as any
2426 detailed proof above the actual definitions of the microarchitectural-flavoured operational semantics have
2427 historically been very resource intensive, up to being the subject of entire Ph.D. theses [6]. Therefore, we
2428 — sadly — defer such formal proof to future work.

2429 In lieu of such formal proof, we compare the models empirically. First, to check for regressions, we ran
2430 the operational model on all the 8950 non-mixed-size tests used for developing the original Flat model
2431 (without instruction fetch or cache maintenance). The results are identical, except for 23 tests which did
2432 not terminate within two hours. We used a 160 hardware-thread POWER9 server to run the tests.

2433 We also ran the axiomatic model on the 90 basic two-thread tests that do not use Arm release/acquire
2434 instructions (not supported by the ISA semantics used for this); the results are all as they should be. This
2435 takes around 30 minutes on 8 cores of a Xeon Gold 6140.

2436 We experimentally tested the equivalence of the operational and axiomatic models on the 52 hand-written
2437 and the 1456 diy-generated tests, checking that the models give the same sets of allowed final states.

2438 6.4 Validating against hardware

2439 To run instruction-fetch tests on hardware, we extended the litmus tool [69]. The most significant extension
2440 consists in handling code that can be modified, and thus has to be restored between experiments. To that
2441 end, we make litmus execute copies of the code, which reside in `mmap`'d memory with execute permission
2442 granted. Copies are made from 'master' copies, which are, in effect, C functions whose contents consist
2443 of gcc extended inline assembly. Of course, such code has to be position independent, and explicit code
2444 addresses in test initialisation sections (such as in `0:X1=1` in the test of §3.3) are specific to each copy. All
2445 the cache handling instructions used in our experiments are all allowed to execute at exception level 0
2446 (user-mode), and therefore no additional privilege is needed to run the tests.

2447 6.4.1 Results from hardware

2448 We ran both the hand-written instruction-fetch litmus tests, and the 1456 auto-generated ones, on a
2449 variety of hardware implementations. A short table of results can be found in Figure 6.2.

2450 Our testing revealed a hardware bug in the Qualcomm Kryo cores of the Snapdragon 820: an illegal
2451 outcome was exhibited by `MP.RF+cachesync+ctrl-isb` (Figure 3.11, p.51) in 84/1.1G runs (not shown
2452 in the table), which we have reported. We have also observed an anomaly for `MP.FF+cachesync+fpo`
2453 (Figure 3.13, p.52) on an Arm-designed core, although this core had (in previous work) been discovered to
2454 suffer a read/read coherence violation. Apart from these, the hardware observations are all allowed by our
2455 models.

2456 As is expected, hardware does not make full use of the architectural envelope: some tests are architecturally
2457 allowed, but never observed on hardware. There are broadly two reasons why this may be: the architecture
2458 must be robust to future designs one might imagine but are not typically implemented (such as the
2459 direct-data-intervention explanation of `SM.F+ic` (Figure 3.19, p.55)); in other cases, the architects must
2460 make decisions to avoid ambiguity in the model, and where unconstrained by hardware designs or software
2461 requirements we choose the simpler programming model (as in `FOV` (Figure 3.20, p.56)).

2462 We therefore draw high confidence that the presented models correctly capture the architectural intent,
2463 and are consistent with existing hardware. There were no existing hardware with either cache type
2464 strengthening at the time of the work, and so, while we believe the models consistent with the architectural
2465 intent, we are unable to give the same level of confidence in those aspects of the model. However, overall
2466 we believe the models are strong enough to forbid the key behaviours guaranteed by hardware, and relied
2467 on by software, while still being loose enough to be consistent with expected potential future designs.

Test	Arch. Intent	H/W Obs.
CoFF	allow	42.6k/13G
CoFR	forbid	0/13G
CoRF+ctrl-isb	allow	3.02G/13G
SM	allow	25.8G/25.9G
SM+cachesync-isb	forbid	0/25.9G
MP.RF+dmb+ctrl-isb	allow	480M/6.36G
MP.RF+cachesync+ctrl-isb	forbid	0/13G
MP.FR+dmb+fpo-fe	forbid	0/13G
MP.FF+dmb+fpo	allow	447M/13G
MP.FF+cachesync+fpo	forbid	F 2.3k/13G
ISA2.F+dc+ic+ctrl-isb	forbid	0/6.98G
SM.F+ic	allow	^U 0/12.9G
FOW	allow	^U 0/7G
MP.RF+dc+ctrl-isb-isb	allow	^U 0/12.94G
MP.R.RF+addr-cachesync+dmb+ctrl-isb	forbid	0/6.97G
MP.RF+dmb+addr-cachesync	allow	^U 0/6.34G

Figure 6.2: Instruction-fetch hardware results

The hardware observations are the sum of testing seven devices: a Snapdragon 810 (4x Arm A53 + 4x Arm A57 cores), Tegra K1 (2x NVIDIA Denver cores), Snapdragon 820 (4x Qualcomm Kryo cores), Exynos 8895 (4x Arm A53 + 4x Samsung Mongoose 2 cores), Snapdragon 425 (4x Arm A53), Amlogic 905 (4x Arm A53 cores), and Amlogic 922X (4x Arm A73 + 2x Arm A53 cores). **U**: allowed but unobserved. **F**: forbidden but observed.

Virtual memory

2468
2469
2470 *This part is based on: Relaxed virtual memory in Armv8-A [37] by Ben Simner, Alasdair Armstrong,*
2471 *Jean Pichon-Pharabod, Christopher Pulte, Richard Grisenthwaite, and Peter Sewell, published in the*
2472 *proceedings of the 31st European Symposium on Programming (ESOP, 2022). The summary of the Arm*
2473 *virtual memory architecture is based on Chapter D5 of the Arm Architecture Reference Manual DDI*
2474 *0487H.a [12].*

Pagetables and the VMSA

7.1 Introduction

Modern computers heavily rely on *virtual memory* to enforce security boundaries: hypervisors and operating systems manage mappings from virtual to physical addresses in order to restrict the access individual processes and guest operating systems have to the underlying physical memory, and to memory-mapped devices. With the endemic use of memory-unsafe languages, even for critical infrastructure, understanding and verifying the programs which manage virtual memory mappings is more vital than ever, driving current interests in hypervisors.

This chapter continues with a brief overview of Arm’s virtual memory systems architecture, in enough detail to understand the subsequent chapters, but does not present new contributions or novel research. The rest of this part then describes the relaxed behaviours of virtual memory ([Chapter 8](#)), gives an axiomatic semantics that capture these behaviours ([Chapter 9](#)), and validates that model against hardware and the architectural intent ([Chapter 10](#)).

7.2 Virtual Memory

Arm’s *virtual memory system architecture* (*VMSA*) defines the virtual memory and virtualisation features of the Arm architecture. It is described, in detail, in the Arm Architecture Reference Manual [81].

Conventionally, memory is imagined as a flat array of bytes, indexed by *physical addresses*. Larger ‘application’ class processors rely heavily on virtual memory: interposing one or more layers of indirection between the accesses of a program (using *virtual* addresses) and the ‘true’ physical addresses of memory. This indirection allows systems running on those processors to:

1. Partition the physical resources between different programs, giving access to only those resources that each program needs, and protecting those resources from other programs that do not need to access them;
2. Indirect accesses through specific ranges of addresses with convenient numeric values or different permissions e.g. to obfuscate the true allocation of resources or to split permissions of a resource for compartmentalisation.
3. Update those indirections at runtime to add, remove, or otherwise modify, the mappings to physical memory, to support techniques such as copy-on-write and paging.

Typically, operating systems split individual programs into distinct *processes*, where each process is associated with its own virtual to physical mapping. Such a mapping corresponds to a partial function, from that process’s own (virtual) addresses to the real hardware physical addresses, with some permissions:

$$\text{translate} : \text{VirtualAddress} \rightarrow \text{PhysicalAddress} \times 2^{\{\text{Read,Write,Execute}\}}$$

Note that this is a simplification. See [The Arm translation table walk \(§7.4\)](#) for a more detailed description of the access permissions, memory types, and other attributes.

2509 Typically operating systems create one such mapping for each process, thereby partitioning the physical
 2510 memory into distinct subsets of physical addresses (which become the *range* of the translate function),
 2511 and would allocate some convenient numeric values to be the virtual addresses the process interacts with
 2512 (which become the *domain* of the translate function). Having this separation allows the processes to be
 2513 given conveniently aligned contiguous chunks of virtual address space even if the underlying physical
 2514 resources are highly fragmented, or, in the case of paging, perhaps not present at all. Additionally,
 2515 operating systems can provide many processes with mappings to the same physical resource (such as
 2516 memory-mapped devices) and control which processes have access to such devices at any point in time.

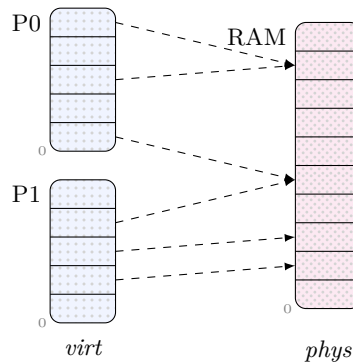


Figure 7.1: Example virtual and physical address spaces for two processes.

2517 The mapping defines an *address space*: the range of virtual addresses a program has access to, and what
 2518 they correspond to. The diagram in Figure 7.1 illustrates an example for two processes. The diagram
 2519 represents the mappings:

- 2520 ▷ For P0:
 - 2521 – virtual addresses in pages 1, and 3 are unmapped.
 - 2522 – virtual addresses in page 2 and 4 map to physical addresses in physical page 8.
 - 2523 – virtual addresses in page 0 map to physical addresses in physical page 5.
- 2524 ▷ For P1:
 - 2525 – virtual addresses in pages 0 and 4 are unmapped.
 - 2526 – virtual addresses in page 1 map to physical addresses in physical page 1.
 - 2527 – virtual addresses in page 2 map to physical addresses in physical page 2.
 - 2528 – virtual addresses in page 3 map to physical addresses in physical page 4.

2529 For example, with a page size of 4k, if process P0 reads or writes the address 0x2305 it will actually access
 2530 physical location 0x8305, since virtual page 2 was mapped to physical page 8 in P0's address space, and
 2531 the offset within a page is preserved.

2532 Each address space corresponds to a distinct translation function. These mappings may be: non-injective
 2533 (contain *aliasing* of multiple virtual addresses to the same physical address); partial (where some virtual
 2534 addresses do not map to a physical address at all); or overlapping with other processes' address spaces, in
 2535 either the domain or the range or both.

2536 Large application-class processor architectures provide hardware support in the form of the *memory*
 2537 *management unit* (MMU), which, once configured by software, will perform the translation from virtual to
 2538 physical addresses and any checking of permissions automatically. Software then needs only manage a set
 2539 of translation functions, in whichever encoding the architecture prescribes (see §7.3 for the encoding used
 2540 by Arm), switch between translation functions on a context switch, and handle any processor exceptions
 2541 generated by the MMU.

2542 **7.3 Arm Translation Tables**

2543 On Arm, as with most architectures, software can configure the MMU through the creation and modification
 2544 of sets of *translation tables* (also referred to as *page tables*).

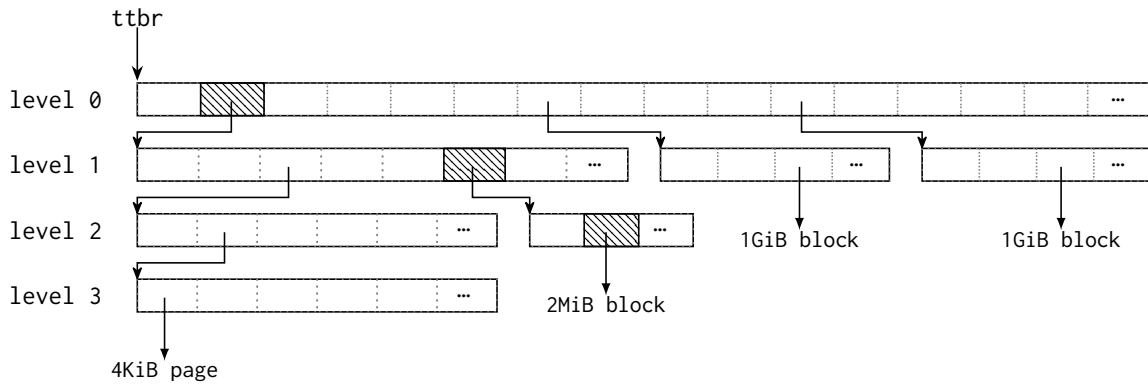


Figure 7.2: Schematic view of an example tree of translation tables. There are seven individual translation tables, over four levels, which defines an address space that maps four separate spans of virtual addresses to spans of physical addresses. In this example, the 2 megabyte block at level 2 encodes the mapping – the output address, permissions, and memory type – for addresses in the range $0x8140200000$ up to $0x81403fffff$ inclusive, which is determined from the highlighted path in the tree: it is the second level 2 (2M span) entry, for the 6th level 1 (1G span) entry, for the second level 0 (512G span) entry, from the root.

2545 The translation tables form an in-memory tree data structure which encode a translation function. Software
 2546 creates and maintains these trees, and controls which tree the MMU uses at runtime. On each memory
 2547 access, the hardware reads from this tree structure to perform the translation, or from one of the various
 2548 caching structures (described in §7.7).

2549 A pointer to the root of the tree is stored in a TTBR (“Translation table base register”), which is one of a
 2550 family of related registers (see §7.6) that determines which tree of translation tables is currently in use by
 2551 that processor.

2552 Each node in the tree is a page-aligned chunk of memory whose interpretation is an array of 64-bit entries,
 2553 where each entry controls the mapping for a particular span of the domain, defining whether the virtual
 2554 addresses in that span are defined for that process, and, if so, what the output physical address is and
 2555 what permissions the process has for that memory. The root table controls the entire address space. The
 2556 tree may recursively split spans into sub-trees. The width of the span mapped by each entry depends on
 2557 its ‘level’, which increases with depth. Typically, the root is at level 0, and the tree has maximum depth
 2558 of 4 (up to level 3) with a page size of 4 KiB. Thus, each pagetable contains 512 entries, with entries
 2559 in the root table each corresponding to a 512 GiB span. Note that Arm is highly configurable and this
 2560 merely represents one common configuration.

2561 Figure 7.2 shows a view of an example set of translation tables, with four mapped regions defined in a tree
 2562 of seven tables. Each rectangular array represents one contiguous page-aligned block of memory, made
 2563 up of 512 64-bit entries. The base register points to the start of the level 0 table (the ‘root’ table). The
 2564 second, seventh, and eleventh, indexes in the root table contain pointers to subsequent (level 1) tables,
 2565 and so on. The exact format of these entries is described in the next section (see §7.3.1).

2566 7.3.1 Translation table format

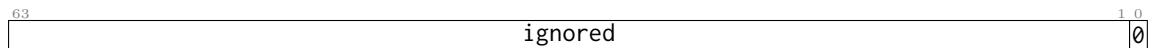
2567 Arm’s virtual memory system architecture is highly configurable. Writing to the SCTLR (“System control
 2568 register”) and TCR (“Translation control register”) system registers allow the programmer to configure the
 2569 processor with a variety of options. To give just a flavour of this configurability, some of those options
 2570 include: the size of virtual addresses; the number of levels in the tree; the starting level; the size of a single
 2571 page (or in Arm terminology, the size of the *translation granule*); the number of address space identifiers
 2572 (ASIDs and VMIDs, used for indexing the caches, see §7.7); alignment requirements; memory attributes
 2573 for hardware walks; enabling hardware management of access flags and dirty bits; write-execute-never
 2574 permissions; and so on.

2575 To simplify the examples we suppress this complexity, and concretise down to just one common instance:
 2576 the one currently used by the Linux kernel; this gives a tree of translation tables with maximum depth 4,
 2577 with 4KiB pages, and 48-bit addresses. In this configuration, each node is a table of 512 64-bit entries, in
 2578 one 4096-byte block of memory.

2579 Each page table entry is one of:

- 2580 1. An *invalid* entry, which indicates that this slice of the domain is unmapped.
- 2581 2. A *table* entry, pointing to a next-level table (a child tree) which recursively maps this slice of the
 2582 domain.
- 2583 3. A *page* (last-level) or *block* (non-last-level) entry which defines a single fixed-size mapping for this
 2584 slice of the domain.

2585 **Invalid entries** An invalid entry is defined by the least-significant bit of the entry being 0. The top 63
 2586 bits of an invalid entry are ignored by hardware, and software is free to use those bits to store metadata.
 2587 Invalid entries may exist at any level in the tree.



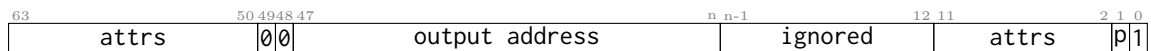
2589 **Block or page entries** Block and page entries are similar to each other: both create a mapping for a
 2590 contiguous slice of the domain mapped by the entry, encoded as an output address (OA) of the base address
 2591 of the slice, with some metadata describing access permissions, memory type, and some software-defined
 2592 bits.

2593 The OA is aligned to the size of the slice of the domain being mapped. For page entries, the OA is aligned
 2594 on a page boundary. A block entry’s OA at level 2 would be 2MiB aligned, and a block entry’s OA at
 2595 level 1 would be GiB aligned. This corresponds to the hardware reserving bits[n:12] of the entry to be
 2596 0 depending on how deep the entry is: at level 1, n=30; at level 2, n=21; and at level 3, n=12.

2597 Block entries can exist at levels 1 and 2. Page entries can only exist at level 3.

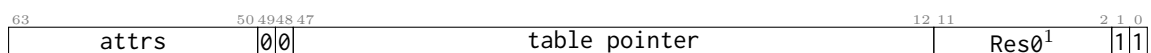
2598 For block entries, bit[1] is 0, for page entries, bit[1] is 1.

2599 Metadata (access permissions, shareability, memory type) are encoded into the *attrs* bits, described more
 2600 in §7.3.2.



2602 **Table entries** A table entry contains a page-aligned pointer to a child table, but can also contain
 2603 similar metadata as the block or page entry, including access permissions (read/write/execute), which are
 2604 combined with any permissions from the child table.

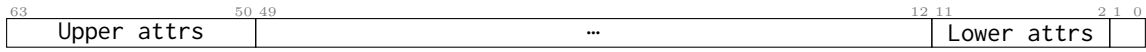
2605 Table entries are allowed only at levels 0–2.



¹The Arm architecture requires these bits are 0 and are reserved for future use.

7.3.2 Attributes

The encoding of the attributes are split into upper and lower attribute fields:



These fields can be further split (see the Arm ARM D8.3.2 for a more comprehensive breakdown) [81]:

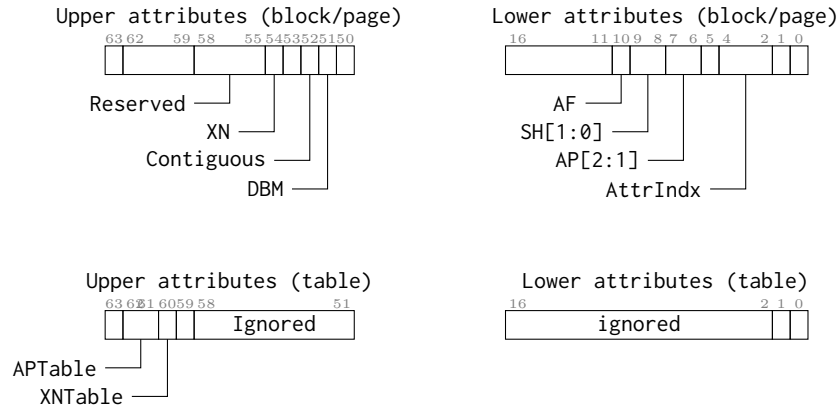


Figure 7.3: Upper and lower attribute encodings for Stage 1 pagetable entries for the 4KiB granule.

Some fields are elided, either because they are for out-of-scope features or otherwise uninteresting, leaving just the following fields of interest:

- ▷ **XN/XNTable:** Execute-Never; when set, this mapping (or child mappings if XNTable) does not have execute permissions.
- ▷ **Contiguous:** allows software to inform hardware that a sequence of entries point to contiguous blocks of output memory, to enable more efficient TLB packing.
- ▷ **DBM/AF:** Dirty bit modifier and access flag; these bits allow software to monitor accesses to locations, which are out-of-scope for this work.
- ▷ **SH:** Shareability; how ‘far’ into the system the memory must be kept coherent for, e.g. memory marked non-shareable need not be coherent for multiple cores. We do not model shareability domains here, so always assume ‘Inner Shareable’.
- ▷ **AP/APTable:** Access permissions; described below in ‘[Access permissions](#)’.
- ▷ **AttrIndx:** Memory attribute; described below in ‘[Memory Attributes](#)’.

Access permissions

Once the walk is complete, and the final output address calculated, the MMU checks to see whether the requested access is permitted. Each level of the table can contain some access permissions which are combined at the end to calculate the final permissions.

For data accesses (reading and writing), table entries have an APTable field (bits[62:61]), and block/page entries have an AP[2:1]¹ field (bits[7:6]). These fields can be decoded using the following table:

Field	When set (1)	When unset (0)
AP[2]	Read-only	Read&Write
AP[1]	Allow at EL1&0	Allow at EL1 only
APTable[1]	Force read-only	No effect on permissions.
APTable[0]	Force forbid access at EL0	No effect on EL0 permissions.

For executable permissions, which permit or forbid instruction fetching from some region of memory, there are no dedicated encodings of the access permission bits. Instead, all mappings are executable by

¹Block/page entries do not store the entire AP field but only AP[2:1]. AP[0] is not present in AArch64.

APTable[1:0]	AP[2]	AP[1]	EL1			EL0		
			R	W	X	R	W	X
			00	0	0	✓	✓	✓
	0	1	✓	✓	×	✓	✓	✓
	1	0	✓	×	×	×	×	✓
	1	1	✓	×	✓	✓	×	×
01	0	0	✓	✓	✓	×	×	✓
	0	1	✓	✓	×	× [†]	× [†]	✓
	1	0	✓	×	×	×	×	✓
	1	1	✓	×	✓	× [†]	×	×
10	0	0	✓	× [†]	✓	×	×	✓
	0	1	✓	× [†]	×	✓	× [†]	✓
	1	0	✓	×	×	×	×	✓
	1	1	✓	×	✓	✓	×	×
11	0	0	✓	× [†]	✓	×	×	✓
	0	1	✓	× [†]	×	× [†]	× [†]	✓
	1	0	✓	×	×	×	×	✓
	1	1	✓	×	✓	× [†]	×	×

Figure 7.4: Merging Access Permissions (Stage 1, EL1&0).
 Entries with a † highlight differences from the APTable=00.

2633 default, unless one of the following applies: the region is mapped writeable at EL0, as writeable EL0
 2634 regions are never executable at EL1; a global WXN (‘Write-execute-never’) configuration bit is set, and the
 2635 entry was writeable; or, when one of the various translation table entry XN (‘Execute-never’) bits are set.
 2636 For simplicity, we assume the execute-never bits are always disabled.

2637 To combine access permissions from the whole walk, the MMU takes the bitwise union of each of the
 2638 APTable fields from each table entry, and then intersects the result with the final AP[2:1] field to produce
 2639 a final set of permissions. Figure 7.4 contains a decoding table for a given table and leaf access permissions,
 2640 for testing whether a requested access is permitted. If the requested access is not permitted, then the
 2641 MMU generates a permission fault, which is reported back to the processor.

2642 Memory Attributes

2643 The processor does not know what is located at any physical address. It may be dynamic random-
 2644 access memory (DRAM, what one would generally consider ‘memory’), but there may also be other
 2645 memory-mapped devices, non-volatile memory, other peripherals, or possibly nothing at all.

2646 To tell the hardware, and to prevent it from performing unsafe optimisations such as speculatively
 2647 attempting to read from a device, software must mark regions of memory as one of either *device* memory,
 2648 *normal cacheable* memory, or normal *non-cacheable* memory, using the translation tables.

2649 The desired memory type is determined from the AttrIndx field (bits[4:2]) in block and page entries.
 2650 Instead of being directly encoded into this field, Arm chose to have the actual attributes stored in a
 2651 separate register: the MAIR (‘Memory attribute indirection register’) register. The MAIR stores an array of
 2652 eight 8-bit fields each of which contains an encoding of a memory type. The AttrIndx field in the entry is
 2653 an integer in the range 0–7, which is used as to index the fields in the MAIR register.

2654 This indirection means that the final result of translation depends not only on the value of the final leaf
 2655 entry in memory, but on the value of certain system registers, such as the MAIR.

2656 Below are the three most common encodings for a MAIR field, and the ones that will be useful later when

2657 discussing tests:

- 2658 ▷ 0b0000_0000: device memory.
- 2659 ▷ 0b0100_0100: normal non-cacheable memory.
- 2660 ▷ 0b1111_1111: normal cacheable memory, inner&outer write-back non-transient, read&write-allocating.

2661 Memory locations marked as device tell the hardware that reads or writes to those locations may have
2662 side-effects. This means hardware treats those locations differently: there will be no speculative instruction
2663 fetches, reads, or writes to those locations; writes to those locations will not *gather* into larger writes;
2664 reads and writes to those locations will not re-order with respect to others; those locations generally will
2665 not get cached; and other thread-local optimizations get disabled. Note that Arm define a wide range of
2666 device memory types, allowing the systems programmer to selectively re-enable some of the previously
2667 described behaviours to enable better performance where they deem it safe to do so.

2668 For normal memory, the software can choose between *cacheable* or non-*cacheable* memory. Arm provide a
2669 range of different options for the cacheability:

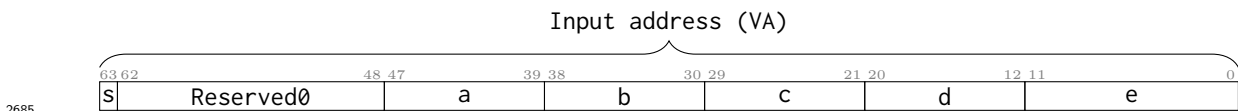
- 2670 ▷ non-cacheable
- 2671 ▷ write-back cacheable
- 2672 ▷ write-through cacheable

2673 As with other features, there is a wide scope for configuration: separately configuring inner (L1, L2) and
2674 outer (L3) caches, and adding cache allocation hints (allocating on reads, writes or both).

2675 7.4 The Arm translation table walk

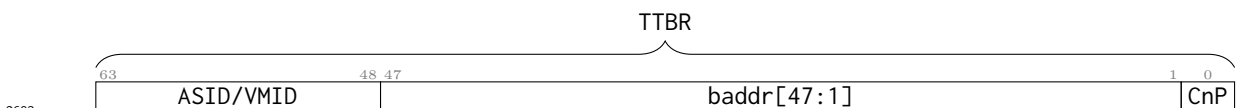
2676 When the processor executes an instruction which takes an address, such as a load or store, the (virtual)
2677 address is converted to a physical address by the MMU. Without a previously cached result (see §7.7),
2678 the MMU must perform a hardware translation table walk.

2679 To do this walk, the MMU reads the relevant TTBR to get the currently in-use tree of translation tables,
2680 and performs a walk of the tree. The hardware walker first slices up the input virtual address into chunks:
2681 the most-significant bit (the sign) is used to determine which base register to use (see §7.6); the next 15
2682 bits are required to be zero; and the rest of the address is split into 9-bit fields which here we call *a*—*d*,
2683 with the remaining bits as field *e*. Fields *a*—*d* are used for indexing into the tables; and field *e* is the
2684 offset in the page, which is always preserved.



2686 Figure 7.5 gives a simplified algorithm for the hardware walk the MMU does on Arm-A, fixed to the
2687 configuration we consider here, eliding the permissions check and hierarchical attribute calculations.

2688 **Reading the TTBR** The base address register contains three fields: the higher bits store the ASID (see
2689 §7.7), or the VMID if for the second stage of a two-stage regime (see §7.5, §7.6); bits 47-1 contain bits 47-1
2690 of the physical address of the root of the translation tables; the final bit is the ‘Common not Private’
2691 (CnP) bit, which is used to indicate when a cluster of processors share the same address space and base
2692 address, which enables further performance optimisations.



```

1: procedure TRANSLATEADDRESS(VA, isRWX)           ▷ Input address, and access kind (read/write/execute)
2:   t ← READ_TTBR().base_address                 ▷ See §7.6, and Reading the TTBR below
3:   attrs ← 0
4:   for i = 0, ..., 3 do                         ▷ Iterate down the levels of the tree
5:     s ← BITSlice(VA, 47 - 9i, 47 - 9i - 9 + 1)   ▷ Slice out fields a—d depending on index
6:     entry ← Mem[t + 8s]                          ▷ Access entry in table
7:     if entry[0] = 0 then                        ▷ Invalid entry
8:       return TRANSLATIONFAULT(VA, Invalid)      ▷ See Faults below
9:     else if entry[1] = 1 ∧ i < 3 then          ▷ Table entry
10:      t ← entry.table_pointer
11:      attrs ← attrs | entry.attrs
12:     else if entry[1] = 0 ∧ (i = 0 ∨ i = 3) then
13:       return TRANSLATIONFAULT(VA, Reserved encoding)
14:     else                                       ▷ Block/page entry
15:       attrs ← attrs | entry.attrs
16:       offset ← BITSlice(VA, 47 - 9i - 9, 0)
17:       OA ← entry.output_address :: offset       ▷ See Computing the final output address below
18:       if !CHECKPERMISSIONS(attrs, isRWX) then  ▷ See §7.3.2 ‘Access permissions’ above
19:         return TRANSLATIONFAULT(VA, Permission error)
20:       else
21:         return OA
22:       end if
23:     end if
24:   end for
25: end procedure

```

Figure 7.5: Simplified single-stage translation table walk for a 4K pagetable.

2694 **Computing the final output address** The output address (OA) of the final descriptor is the start of the
2695 range mapped by the entry. The offset into the range must be added to the start, in order to compute the
2696 final output address of the translation.

2697 To compute this address, the MMU takes the OA field from the entry, and the level in the tree the entry
2698 is at, and ‘completes’ the address by bitwise appending the remaining fields to create the complete 48-bit
2699 output address. Recall that the OA field of the block mappings gets wider the deeper in the tree you are,
2700 and so for a 1GiB entry the OA field is only 18 bits wide, but for a 4KiB page entry its OA field is the full
2701 36 bits.

2702 ▷ For a 1GiB (level 1) block entry; PA = OA::c::d::e

2703 ▷ For a 2MiB (level 2) block entry; PA = OA::d::e

2704 ▷ For a 4KiB (level 3) page entry; PA = OA::e

2705 Note that this process means that the least-significant 12 bits of the input VA are unchanged and remain
2706 the same in the final output PA, regardless of how the translation function is configured.

2707 **Faults** The MMU may emit one of several fault types during a translation table walk (these are referred
2708 to by Arm as the *MMU fault* types):

2709 ▷ Translation fault.

2710 These are generated when the mapping in the translation table is invalid, either because bit[0]
2711 was 0, or because the descriptor encoding was reserved-as-invalid. Translation faults also result
2712 from trying to translate an address that is outside the 48-bit input address range (i.e. the bits
2713 reserved-as-zero in the address were set).

2714 ▷ Permission fault.

2715 Generated when the mapping was valid, but the access permissions do not permit the requested
2716 access (for example, trying to write to a read-only address).

2717 ▷ Access flag fault.

2718 These are generated when hardware management of access flags is disabled and the access flag bit is
2719 set.

2720 ▷ TLB Conflict aborts.

- 2721 ▷ Alignment fault.
- 2722 Generated when an operation requires an aligned memory address, but is given a misaligned one.
- 2723 ▷ Address size fault.
- 2724 Generated when the OA, or TTBR, has a value that is out of the physical address range.
- 2725 ▷ Synchronous external abort on a translation table walk.
- 2726 These are *external aborts* (that come from the system not from the MMU) that happen due to
- 2727 accesses that the MMU generated. For example, if the next-level table field pointed to an address
- 2728 for which there was no memory or device, the system-on-chip would return a fault to the processor.

2729 These faults lead to processor exceptions. The fault type is stored in the ESR (“exception syndrome
 2730 register”) register, in its EC (“exception class”) field, and any supplementary information is stored in its
 2731 ISS (“instruction specific syndrome”) field (such as which level in the tree the fault came from, whether
 2732 the originating instruction was a read or a write, and so on). Exception handling code can read the
 2733 ESR register to determine the fault type and cause, and can read the FAR (“fault address register”) to
 2734 determine the virtual address which triggered the fault, and handle the fault appropriately.

2735 **7.5 Virtualisation**

2736 So far, this chapter has focused on operating systems and processes. However, modern systems isolate
 2737 not just processes within an operating system, but entire operating systems from one another within a
 2738 hypervisor.

2739 To achieve, hardware adds another layer of virtual memory, in addition to the existing one, creating
 2740 two *stages* of translation. Processes use virtual addresses, which are converted to *intermediate physical*
 2741 addresses (IPAs, also sometimes known as *guest-physical* addresses) using the operating system’s configured
 2742 translation tables. These then go through another *stage* of translation, typically controlled by the
 2743 hypervisor, converting those IPAs into physical addresses.

2744 Software manages both sets of translation tables: operating systems manage *Stage 1* tables to convert VAs
 2745 to IPAs; and hypervisors manage *Stage 2* tables to convert those IPAs to PAs. This gives two separate
 2746 translation functions, which the hardware composes together at runtime:

$$\begin{aligned} \text{translate_stage1} &: \text{VirtualAddress} \rightarrow \text{IPA} \times \text{Permissions} \times \text{MemoryType} \\ \text{translate_stage2} &: \text{IPA} \rightarrow \text{PhysicalAddress} \times \text{Permissions} \times \text{MemoryType} \end{aligned}$$

2747 Hypervisors (running at EL2) configure the second-stage translation in much the same way as operating
 2748 systems configure the first stage: by creating a tree of translation tables, with an almost identical format
 2749 as before, and storing a pointer to the root of this tree in the VTTBR (“Virtualization translation table
 2750 base register”). The hardware reads the VTTBR to perform a second-stage translation to convert an IPA to
 2751 a PA, and will do the translation table walk over that tree in much the same way as described earlier for
 2752 (what we can now call) the first-stage translation.

2753 This results in two address spaces, a virtual address space and an intermediate-physical address space.
 2754 Figure 7.6 contains an example layout of these address spaces for a machine running three processes (P0,
 2755 P1, P2) in two operating systems (OS0, OS1). As with the earlier diagram in Figure 7.1, each column is a
 2756 (set of) address spaces, with transformations between them defined by their respective translation functions.
 2757 On the left-hand side are the virtual address spaces of the various processes, whose virtual addresses
 2758 are translated (using the translation tables pointed to by the TTBR register) into intermediate-physical
 2759 addresses in the central address spaces (for the respective OS). Those IPAs are then translated (using the
 2760 VTTBR) into physical addresses.

2761 Concretely, if P1 reads from address $0x1001$, it will be translated into the IPA $0x3001$, in OS0’s address
 2762 space. This IPA is then translated again into the physical address $0x6001$, by a second stage of
 2763 translation controlled by the hypervisor, and the processor will actually read from the RAM at location
 2764 $0x6001$.

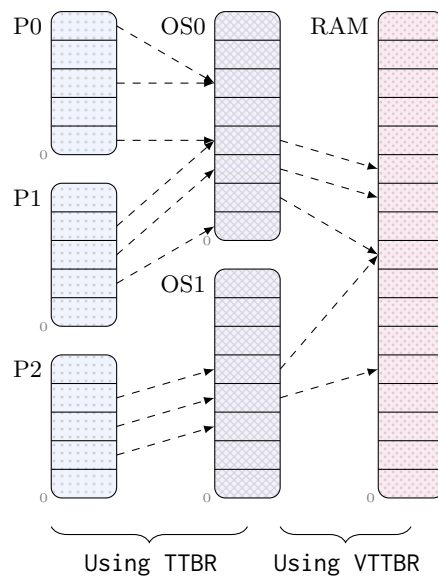


Figure 7.6: Example virtual, intermediate physical, and physical address spaces for three processes running on two operating systems.

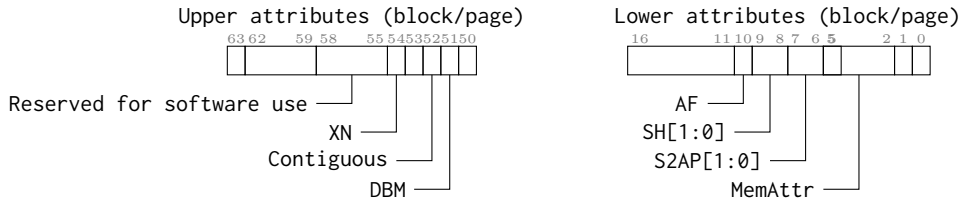


Figure 7.7: Attribute encodings for Stage 2 pagetable entries for the 4KiB granule [81, D8.3].

Field	When set (1)	When unset (0)
S2AP[1]	Writeable	not Writeable
S2AP[0]	Readable	not Readable

Figure 7.8: S2AP field encoding.

Stage 2 attributes encoding Stage 2 translation table entries are encoded similar to their stage 1 counterparts, as illustrated in Figure 7.7, with some minor differences:

- ▷ Stage 2 table entries do not have any additional attributes, and so do not have an APTable field.
- ▷ The Stage 2 AP field (called S2AP) has a slightly different (and simpler) format, see Figure 7.8.
- ▷ Stage 2 block and page entries do not have a MemAttrIndx field but rather encode the memory type directly into the MemAttr field bits[5:2] (see the full description in the Arm ARM [12, D5-4874] for all possible encodings):
 - 0b0000: Device memory.
 - 0b0101: Normal non-cacheable.
 - 0b1111: Normal write-back inner&outer cacheable.

These are interesting as they mean that the stage 1 and stage 2 attributes (permissions and memory types) must be *combined* in order to produce the final output. This combination is not just a case of letting stage 2 overrule the stage 1 settings, but rather that both stages get a veto: if stage 1 sets the memory type to be device or non-cacheable then it overrules what stage 2 sets. Similarly, if stage 1 permissions forbid an access then the stage 2 permissions cannot overrule that.

Second-stage translations during a first-stage walk There is a complication with the story so far. The stage 1 tables are created by the operating system, which is using an intermediate physical address space, not a physical one. The writes the OS does to the tables will be translated, as they are normal data writes. But, the tables themselves contain references to other tables, and those entries will be intermediate physical addresses, and so, they must also be translated, including the value of the TTBR itself.

In our assumed configuration of 4KiB pages and 4 levels of translation, this leads to a maximum of 24 memory accesses to perform the translation: 4 reads of stage 1 translation tables, 16 reads of stage 2 translation tables during those stage 1 walks, and a final 4 reads of the stage 2 translation tables to translate the output IPA into the final PA. Below we will see an example of the accesses performed during a typical store instruction, as illustrated in Figure 7.9.

Figure 7.10 then gives a simplified algorithm for a two-stage translation-table-walk, with some detail elided: the permissions combining and checking, determining current regime, routing of exceptions, and so on. Arm give a full and precise definition of the translation table walk as part of the ASL defining the intra-instruction semantics.

An example Consider the Arm STR Xn,[Xt] instruction. It writes data stored in register Xn to an address stored in register Xt. Figure 7.9 is an example trace of one execution of the aforementioned store instruction. It is just as the Arm intra-instruction semantics would generate when executed at EL0 in the two-stage EL1&0 regime, in the worst case setting where the address is mapped by last level entries, in both the stage 1 and stage 2 pagetables. Each node represents an event in the trace (a memory or register access), and the arrows between them represent control flow within the intra-instruction semantics. The events in the dotted region come from the translation table walk (calls to the Arm AArch64.TranslateAddress pseudocode function).

Translation starts by reading the base address for the stage 1 walk, from the relevant TTBR, and performing

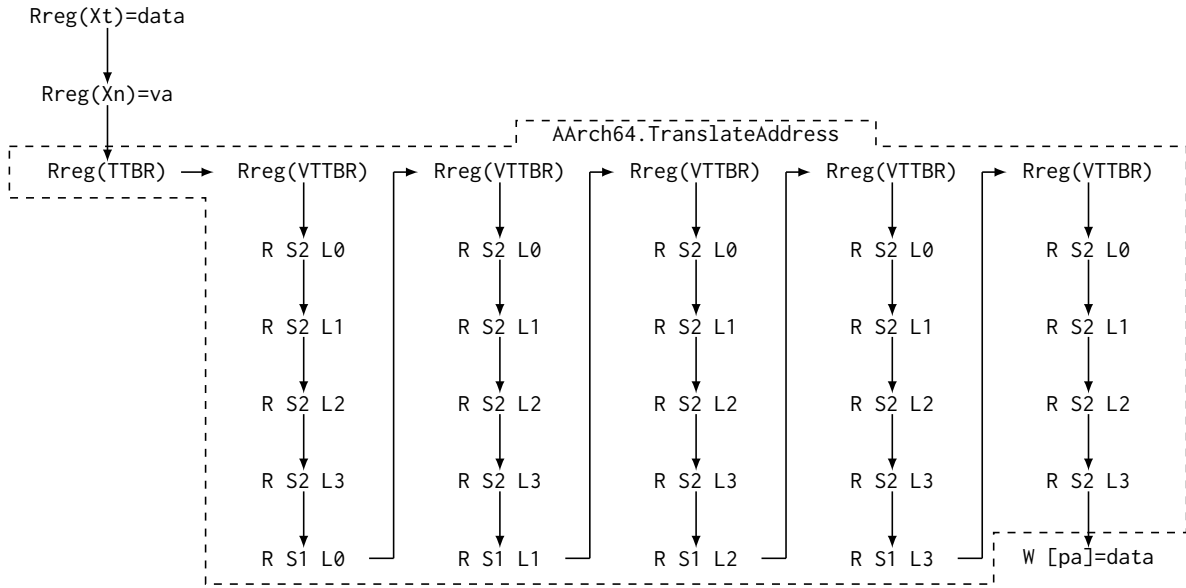


Figure 7.9: Memory and register accesses during a 'STR Xt, [Xn]' instruction.

2803 a second-stage translation (the events marked as R S2 L i) to get the physical address of the stage 0 level 0
 2804 table. It proceeds to read from that table (the event R S1 L0), repeating the process again, once for each
 2805 level in the stage 1 table. Once the final result from the stage 1 walk is obtained (from the event R S1
 2806 L3), the final stage 2 walk is done to calculate the final physical address to be accessed. When the full
 2807 walk is complete, and the pseudocode returns from the walk, it performs the actual memory access (the W
 2808 [pa]=data event in the diagram).

```

1: procedure WALK(Stage, IA, isRWX)                                ▷ IA is now input address, which may be VA or IPA.
2:   if Stage = 1 then
3:     t ← READ_TTBR().base_address                                ▷ See §7.6
4:   else
5:     t ← VTTBR_EL2.base_address
6:   end if
7:   attrs ← 0
8:   for i = 0, ..., 3 do
9:     s ← BITSlice(IA, 47 - 9i, 47 - 9i - 9 + 1)                ▷ Slice out fields a—d depending on level
10:    addr ← t + 8s                                              ▷ Address of entry in the table
11:    if ISINTWOSTAGEREGIME() ∧ Stage = 1 then
12:      addr ← WALK(Stage 2, addr, R)                             ▷ Do a stage 2 walk to get physical address
13:      if addr is TranslationFault then                          ▷ ... which may fail
14:        return TRANSLATIONFAULT(IA, Stage 2 during Stage 1)
15:      end if
16:    end if
17:    entry ← Mem[addr]
18:    if entry[0] = 0 then                                        ▷ Invalid entry
19:      return TRANSLATIONFAULT(IA, Stage, Invalid)
20:    else if entry[1] = 1 ∧ i < 3 then                            ▷ Table entry
21:      t ← entry.table_pointer
22:      attrs ← attrs | entry.attrs
23:    else if entry[1] = 0 ∧ (i = 0 ∨ i = 3) then
24:      return TRANSLATIONFAULT(IA, Stage, Reserved encoding)
25:    else                                                        ▷ Block/page entry
26:      attrs ← attrs | entry.attrs
27:      offset ← BITSlice(IA, 47 - 9i - 9, 0)
28:      OA ← entry.output_address :: offset
29:      if !CHECKPERMISSIONS(Stage, attrs, isRWX) then           ▷ See Stage 2 attributes encoding above
30:        return TRANSLATIONFAULT(IA, Stage, Permission error)
31:      else
32:        return PA
33:      end if
34:    end if
35:  end for
36: end procedure
37:
38: procedure TRANSLATEADDRESS(VA, isRWX)
39:   if ISINSINGLESTAGEREGIME() then
40:     PA_or_Fault ← WALK(Stage 1, VA, isRWX)
41:     return PA_or_Fault
42:   else
43:     IPA_or_Fault ← WALK(Stage 1, VA, isRWX)
44:     if IPA_or_Fault is TranslationFault then
45:       return IPA_or_Fault
46:     end if
47:     IPA ← IPA_or_Fault
48:     PA_or_Fault ← WALK(Stage 2, IPA, isRWX)
49:     return PA_or_Fault
50:   end if
51: end procedure

```

Figure 7.10: Simplified two-stage translation table walk for a 4K pagetable.

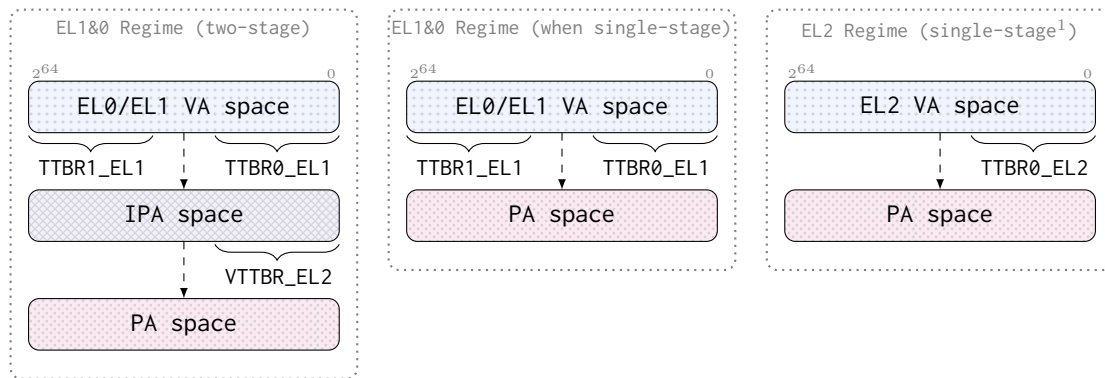


Figure 7.11: Translation regimes that apply to execution at EL0, EL1, and EL2.

7.6 Translation regimes

As mentioned earlier, there are multiple translation table base registers. Each of them defines a translation function, pointing to the root of the tree of translation tables which define it. These translation functions are then composed together into various translation *regimes*, each defining the set of translation functions (and therefore which translation table base registers) which will be used for translations done by the processor.

Arm define a set of these translation regimes. Figure 7.11 gives an overview of three of the most common regimes, which are:

- ▷ EL1&0 (two-stage)
 - For programs executing at EL0 or EL1 when virtualisation is enabled.
 - VAs with the high bit set are translated into IPAs using the EL1-configured register, TTBR1_EL1. VAs are typically split into ‘high’ and ‘low’ regions with different translations, primarily used for separate kernel and user address spaces.
 - VAs without the high bit set are translated into IPAs using the EL1-configured register, TTBR0_EL1.
 - IPAs are translated to PAs using the EL2-configured VTTBR_EL2 register.
- ▷ EL1&0 (single-stage)
 - For programs executing at EL0 or EL1 when virtualisation is disabled.
 - VAs with the high bit set are translated into PAs using the EL1-configured register, TTBR1_EL1.
 - VAs without the high bit set are translated into PAs using the EL1-configured register, TTBR0_EL1.
- ▷ EL2
 - For programs executing at EL2.
 - VAs without the high bit set are translated into PAs using the EL2-configured register, TTBR0_EL2.
 - VAs with the high bit set are always unmapped.

Which translation regime is being used is defined by various system registers and the current system state.

- ▷ Translations at EL1 or EL0 use one of the EL1&0 regimes.
- ▷ Translations at EL2 use the EL2 regime.
- ▷ TCR_EL2 (set at EL2) determines whether the EL1&0 is a single-stage or two-stage regime.
- ▷ TTBR0_EL1, TTBR1_EL1 determine the stage 1 of the EL1&0 regimes, and can only be set at EL1 or higher.
- ▷ TTBR0_EL2 determines the stage 1 of the EL2 regime, and can only be set at EL2 or higher.
- ▷ VTTBR_EL2 determines the stage 2 of the EL1&0 regime, and can only be set at EL2 or higher.

Arm define a wide range of other regimes which we do not cover here, including for EL3, secure mode, and the virtualised host extension (FEAT_VHE); see the Arm ARM [81, §D8.1.2] for more information.

¹EL2 is always a single-stage regime. Note that there is a two-stage EL2&0 regime, which is not discussed here.

7.7 Caching in TLBs

It would have an unacceptable performance penalty to simply perform the (up to) 24 additional memory accesses for every instruction-fetch, read, or write. Therefore, the hardware does not do this. Instead, the results of previous translations of the same address are cached in specialised structures called *Translation Lookaside Buffers* (TLBs). These TLBs can store whole translation results, or the separate virtual and intermediate-physical mappings, or individual translation table entries, or a mix of the above, which we will explore more in the next chapter.

When the processor translates a virtual address, it first looks for it in the TLB. If there is no entry, then this is called a *TLB miss* and a translation table walk must be performed. The results of this walk are typically then cached in the TLB, so future translations of the same address can directly grab the physical address, memory attributes, and permissions, without needing to do another translation table walk. This process and the various microarchitectural structures are explored more in §8.3.1. If there is an entry, this is referred to as a *TLB hit*. In this case, the result can be taken directly from the TLB.

Under normal circumstances, the TLB is invisible to userspace programs. However, unlike normal data caches, TLBs are not kept coherent by the hardware automatically. Therefore, systems code is expected to manage the TLBs manually, using a set of instructions which Arm provide specifically for this purpose: the family of TLBI TLB-maintenance instructions. When context switching, the systems software must manually manage the TLB, invalidating stale entries for old mappings out of the cache. The behaviours that arise from reading from potentially stale TLB entries are explored in detail in §8.5.

Address space identifiers TLB maintenance operations, and the TLB cache misses they subsequently create, impose additional performance penalties on the software using them. To reduce this burden, Arm provide a mechanism to permit multiple processes' address spaces to be loaded into the TLB at the same time, by allowing the software to mark each address space with a numeric label. Arm call these *address space identifiers* (ASIDs), for Stage 1 address spaces, and *virtual machine identifiers* (VMIDs), for Stage 2 address spaces. Entries in the TLB are tagged with the current ASID and VMID, and only that address space will see entries in the TLB with that combination.

The current identifier is encoded in the high-order bits of the current TTBR. During a context switch, the system software needs only switch to the new translation tables for the new address space of the other process. It is not necessary to do TLB maintenance, so long as it ensures the identifiers are distinct.

As there are only finitely many identifiers available (typically it is an 8-bit field), eventually TLB maintenance is required in order to re-use a previously allocated identifier, for a new address space. But, this typically happens far less frequently than context switches between pre-existing address spaces. The provided TLB maintenance instructions can target specific ASIDs or VMIDs, avoiding the need to over-invalidate other cached address space translations, preventing a cascade of TLB misses in other processes, further improving the runtime performance for a small amount of additional effort on the software side.

TLB maintenance Recall that on a TLB miss, any translation table entries read during any hardware translation table walk can be cached in the TLB, tagged with the current context: the translation regime, the ASID and VMID, the VA/IPA being translated, and so on. TLB maintenance instructions remove cached copies of those entries in the TLB.

A single TLB maintenance instruction which cleared the entire TLB would be prohibitively expensive, and often unnecessary; instead, Arm define a large family of instructions for targeted removal of particular entries, under the TLBI mnemonic.

A simplified format for a TLBI mnemonic is a product of fields:

```
1 TLBI <type><regime>{<broadcast>}{,<Xn>}
2
3 <type> =
4 ALL | VMALL | ASID | VA{A}{L} | IPAS2{L}
5 <regime> =
6 E1 | E2
7 <broadcast> =
8 IS
```

2897 The type selects which kinds of cached entries will be affected:

- 2898 ▷ ALL affects all (Stage 1 and 2) entries.
- 2899 ▷ VMALL affects all (Stage 1 and 2) entries, for a particular VMID.
- 2900 ▷ ASID affects all Stage 1 entries, for a particular ASID.
- 2901 ▷ VA affects Stage 1 entries which were cached for translations of the given virtual address. This
- 2902 affects only the ASID provided, or can optionally appended with A, to affect entries for any ASID.
- 2903 Appending L makes the TLB maintenance only affect cached last-level entries (only page, not block).
- 2904 ▷ IPAS2 affects Stage 2 entries which were cached for translations of the given intermediate-physical
- 2905 address, including for cached entries of a Stage 1 walk during a Stage 2 walk. Appending L makes
- 2906 the TLB maintenance only affect cached last-level entries (only page, not block).

2907 The regime selects entries by the tagged translation regime:

- 2908 ▷ E1 only removes cached entries for walks for the EL1&0 regime.
- 2909 ▷ E2 only removes cached entries for the EL2, or the EL2&0 regime, whichever is currently enabled.

2910 Finally, the optional broadcast flag determines which (set) of CPUs the TLB maintenance should affect:

- 2911 ▷ IS (‘Inner-Shareable’) broadcasts the TLB maintenance to all cores in the same Inner Shareable
- 2912 domain, and the TLBI does not finish until all cores have completed their maintenance.

2913 The most common, and the ones that will be discussed in the following chapters, are as follows:

- 2914 ▷ TLBI VAE1, Xn: Invalidate this CPU’s cached copies of entries used to translate the virtual address
- 2915 in register Xn, for the EL1&0 regime, for the current ASID and VMID.
- 2916 ▷ TLBI VALE1, Xn: Invalidate this CPU’s cached copies of any last-level entries used to translate the
- 2917 virtual address in register Xn, for the EL1&0 regime, for the current ASID and VMID.
- 2918 ▷ TLBI VAAE1, Xn: Invalidate this CPU’s cached copies of any entries used to translate the virtual
- 2919 address in register Xn, for the EL1&0 regime, for the current VMID, for any ASID.
- 2920 ▷ TLBI VAE1IS, Xn: Invalidate all CPU’s cached copies of entries used to translate the virtual address
- 2921 in register Xn, for the EL1&0 regime, for the current ASID and VMID.
- 2922 (...and equivalent TLBI VAE2, TLBI VALE2, TLBI VAE2IS instructions for virtual addresses in the
- 2923 EL2 regime)
- 2924 ▷ TLBI IPAS2E1, Xn: Invalidate this CPU’s cached copies of entries used to translate the intermediate
- 2925 physical address in register Xn, for the EL1&0 regime, for the current VMID.
- 2926 ▷ TLBI IPAS2E1IS, Xn: Invalidate all CPU’s cached copies of entries used to translate the intermediate
- 2927 physical address in register Xn, for the EL1&0 regime, for the current VMID.
- 2928 ▷ TLBI VMALLE1: Invalidate this CPU’s cached copies of entries for the EL1&0 regime, for the current
- 2929 VMID.
- 2930 ▷ TLBI VMALLE1IS: Invalidate all CPU’s cached copies of entries for the EL1&0 regime, for the current
- 2931 VMID.
- 2932 ▷ TLBI ALLE1: Invalidate this CPU’s cached copies of entries for the EL1&0 regime, for any ASID or
- 2933 VMID.
- 2934 ▷ TLBI ALLE1IS: Invalidate all CPU’s cached copies of entries for the EL1&0 regime, for any ASID or
- 2935 VMID.
- 2936 (...and equivalent TLBI ALLE2, and TLBI ALLE2IS instructions for the EL2 regime)
- 2937 ▷ TLBI ASIDE1IS, Xn: Invalidate this CPU’s cached copies of entries for the EL1&0 regime, for the
- 2938 ASID specified in register Xn.
- 2939 (Note that the EL2 regime does not have ASIDs)

2940 This is not an exhaustive list, see the full description in the Arm manual for a more complete description

2941 [12, D5-4915], but covers all those that appear in the following chapters.

Relaxed virtual memory

Now, we introduce the main concurrency architecture design questions that arise for virtual memory in Arm. As usual, the architecture defines the *envelope* of behaviours which hardware must guarantee and on which software may rely. This envelope must be tight enough to give the guarantees software needs to function, but still loose enough to admit the range of existing and conceivable microarchitectures whose optimization techniques are necessary for performance.

This chapter discusses both the relevant microarchitecture as we understand it, and also the behaviours which software relies upon. The discussion will touch on points of several kinds: some which are clear in the current Arm prose documentation; some where Arm are in the process of architecting a change; some that are not documented but where the semantics is (perhaps, after discussion with Arm) clear or constrained by current hardware or software practice; and, some where their modelling raised questions for which the architecture is not yet well-defined and Arm must make an architectural decision.

Ideally, we would be able to specify which points belong to which kind. In practice, things are not so simple. In some places, there is no clean separation between aspects there are clearly defined in the architecture reference, and those that are not; the manual sometimes has a shallow covering, with some but not all, of the key details. In other places, the reference may have been updated or changed over the course of the work, clarifying parts of the architecture. While such updates may have happened concurrently with discussions with Arm, the reference text itself is solely the responsibility of Arm. In §8.9 we will return to this, and more directly address the kinds of each point discussed.

Chapter overview The body of this chapter will explore a sequence of key behaviours, some of which the architecture permits, and some that it does not. Each contains a description of the behaviour, including whether software relies on it or known hardware guarantees it; a short discussion of the architectural intent as we understand it; and any associated litmus tests.

This chapter will discuss a variety of interesting behaviours. In an attempt to make this chapter more approachable, it is broken down into a logical progression: slowly building up from the most simple and fundamental parts of the architecture, to increasingly more complex cases.

We first discuss (in §8.2) how translation affects the prior usermode tests covered in previous work, primarily for the case where locations are aliased. Then, we explore how translation entries may be cached (§8.3) and the fundamental behaviours which arise from translation and the walk (§8.4). Building upon that, we will see how those caches affect the discussed behaviours (§8.5). Then, we will explore how the various kinds of TLB maintenance interact with those cached translations (§8.6), and other translation table walks. Finally, we touch on how all of the above fit together with system registers, and other context changing and synchronising operations (§8.7).

2976
2977
2978
2979
2980
2981
2982
2983
2984
2985
2986
2987
2988
2989
2990
2991
2992
2993
2994
2995
2996
2997
2998
2999
3000
3001
3002
3003
3004
3005
3006
3007
3008
3009
3010
3011
3012
3013
3014
3015
3016
3017
3018

Chapter contents

- 8.1 Virtual memory litmus tests 107**
- 8.2 Aliased data memory 109**
 - 8.2.1 Virtual coherence 109
 - 8.2.2 Aliasing different locations 112
 - 8.2.3 Might be same (physical) address 112
 - 8.2.4 Must not be same physical address 112
- 8.3 What can be cached in TLBs 112**
 - 8.3.1 Microarchitectural TLBs 113
 - 8.3.2 Model MMU 114
 - 8.3.3 Invalid entries 115
- 8.4 Reads not from TLB 115**
 - 8.4.1 Out-of-order execution 115
 - 8.4.2 Enforcing thread-local ordering 118
 - 8.4.3 Enhanced Translation Synchronization 123
 - 8.4.4 Forwarding to the translation table walker 124
 - 8.4.5 Speculative execution 125
 - 8.4.6 Single-copy atomicity 127
 - 8.4.7 Multi-copy atomicity 127
 - 8.4.8 Translation-table-walk intra-walk ordering 128
 - 8.4.9 Multiple translations within a single instruction 128
- 8.5 Caching of translations in TLBs 132**
 - 8.5.1 Cached translations 132
 - 8.5.2 TLB fills 133
 - 8.5.3 microTLBs 133
 - 8.5.4 Partial caching of walks 135
- 8.6 TLB maintenance 136**
 - 8.6.1 Recovering coherence 136
 - 8.6.2 Thread-local ordering and TLBI 139
 - 8.6.3 Broadcast 141
 - 8.6.4 Virtualization 145
 - 8.6.5 Break-before-make 148
 - 8.6.6 Access permissions 150
- 8.7 Context synchronisation 153**
 - 8.7.1 Relaxed system registers 153
- 8.8 Problems 154**
 - 8.8.1 Reachability 155
 - 8.8.2 Wide invalidations 155
- 8.9 Contributions 156**
- 8.10 Related work 158**

8.1 Virtual memory litmus tests

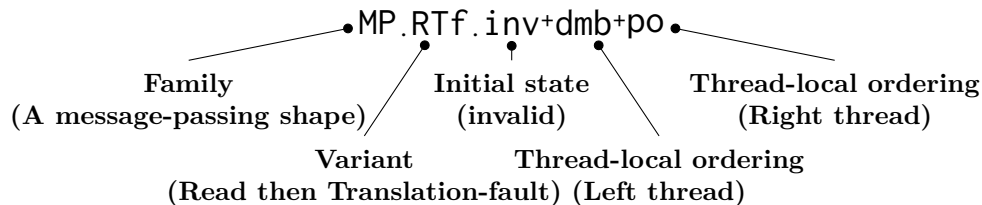
As we explore deeper into the systems semantics, we are exposed to more and more of the microarchitectural machine state; understanding that state is integral to understanding the behaviour of the machine. Virtual memory poses its own specific challenges, but is fundamentally no different than the other fragments of Arm we have seen. As such, exploring the architectural intent is best done through the creation, discussion, and evaluation of, small test programs which are representative examples of common software patterns or interesting hardware behaviours. Therefore, litmus tests exploring those behaviours must include information about not only the memory locations of the test, but also the setup of the pagetables which map them. This is best demonstrated by an example.

A virtual memory litmus test Much as in usermode (and ifetch, see Chapter 3) we examine litmus tests containing a relatively small amount of code corresponding to some interesting behaviour we wish to investigate. To illustrate this, Figure 8.1 contains the test listing for a simple (but non-trivial) virtual-memory litmus test, MP.RTf.inv+dmb+po.

MP.RTf.inv+dmb+po		AArch64
Initial state: x -> invalid, z -> pa1, *pa1 = 1, y -> pa2, 0:X0=desc3(z), 0:X1=pte3(x), 0:X2=1, 0:X3=y, 1:X1=y, 1:X3=x		
Thread 0	Thread 1	Thread1 E11 Handler
STR X0, [X1] DMB ST STR X2, [X3]	LDR X0, [X1] LDR X2, [X3]	MOV X2, #0 MRS X13, ELR_EL1 ADD X13, X13, #4 MSR ELR_EL1, X13 ERET
Allowed: 1:X0 = 1 & 1:X2=0		

Figure 8.1: Test MP.RTf.inv+dmb+po: code listing.

This test is a variant of the classic message-passing test, but where one of the reads in the relaxed cycle of events is an implicit read due to a translation table walk. More specifically, the second read in the right-hand thread is the implicit read of the last-level entry of the stage 1 translation table walk, which in this case was initially invalid and so the interesting executions results in a translation fault. The test is explained in more detail below. In general we can take each of the classic usermode litmus test shapes, and re-imagine them in a virtual memory context, replacing one or more of the explicit memory events in the cycle with implicit ones from one or more translation table walks, and typically making some of the writes be writes to pagetables. We can then assign a relatively lightweight naming scheme for such litmus tests: for example, in MP.RTf.inv+dmb+po, the name can be broken down into separate fields representing the shape (family), which of the events are replaced by implicit ones, and whether the initial state for those implicit accesses are valid or invalid:



Not all litmus tests follow this convention: some do not correspond to a shape from the suite of usermode litmus tests, and others are derived from virtual-memory-specific patterns which arise in software or from discussion with architects.

In detail, this test mimics the usual message-passing pattern, with two locations x and y, with one thread reading the locations sequentially, in the inverse order, reading the 'flag' y first, then the 'data' x second. However, in this case, the data is not a value in a memory location, but the mapping of the memory location itself. This can be seen in the 'Initial State' part of the code listing (Figure 8.1), which contains

3051 not only the usual initial register and memory location values for the test, but also a terse description of
 3052 the initial mappings of those locations: x is invalid, so any access results in a translation fault; z maps to
 3053 physical address $pa1$ which initially holds 1; and y maps to $pa2$, which holds zero. The initial register
 3054 state now also can reference parts of the pagetable: register $X1$ in Thread 0 contains the value $pte3(x)$
 3055 which is the address of the last-level (level 3) entry which is responsible for mapping x ; and $X0$ contains
 3056 the value $desc3(z)$, which is the initial value of the entry responsible for mapping z .

3057 The test then begins in Thread 0 by copying the entry which maps z into the entry which maps x ,
 3058 effectively making x an alias of z , before passing a message to Thread 1 via y . Thread 1 then reads y , and
 3059 then attempts to read x . The second load will either be translated using the new translation, in which
 3060 case it reads from $pa1$ and get 1, or be translated from the initial value and result in translation fault.
 3061 In the case where the second load faults, execution jumps to the ‘Thread 1 EL1 Handler’ block, which
 3062 writes 0 to $X2$ and advances the program counter to the next instruction¹. The final state corresponds
 3063 to an execution in which the first load receives the message, and so reads 1, but the second fails with a
 3064 translation fault reading from a stale translation table entry.

3065 The interesting relaxed execution can be seen as a set of events with some relations which witness the
 3066 order the events happened in. This test’s events diagram is shown in Figure 8.2.

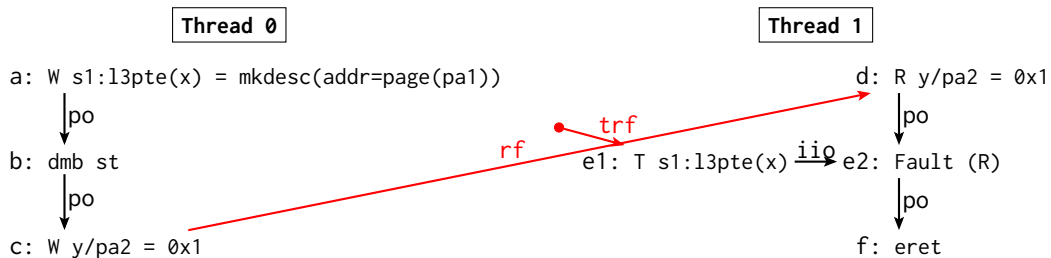


Figure 8.2: Test MP.RTf.inv+dmb+po: execution diagram

3067 These diagrams are much like the ones drawn for usermode tests, but with a few key differences:

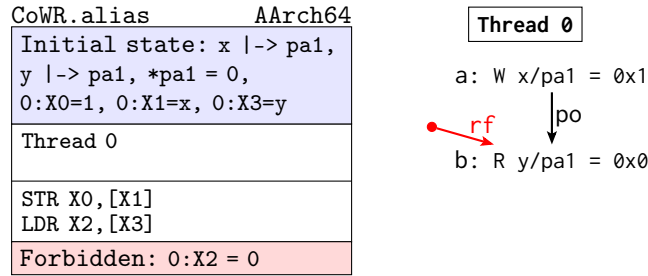
- 3068 ▷ The implicit reads due to translation table walks are included in the execution, labelled with T (for
 3069 Translate), and ordered within an instruction by iio (intra-instruction-order), with each other and
 3070 with the associated explicit events of the instruction.
- 3071 ▷ Memory accesses are annotated with both their virtual and physical addresses, e.g. event d: R
 3072 $y/pa2 = 0x1$ denotes the read for a virtual address y , and read from the physical address $pa2$.
- 3073 ▷ We introduce a notation whereby some addresses and values can be written using symbolic func-
 3074 tions, e.g. a: W $s1:l3pte(x) = mkdesc(addr=page(pa1))$ denotes the write is to the stage 1 level 3
 3075 pagetable entry which maps x , with a value that is a 64-bit descriptor whose output-address field is
 3076 for $pa1$ ’s page.
- 3077 ▷ Accesses which fault generate a Fault event, annotated with the access kind (read/write/execute).

3078 We elide translation read events, physical address labels, and other uninteresting and extraneous details.

3079 **Symbolic helper functions** These symbolic functions are implemented as part of the `isla-cat` language,
 3080 accepted by `isla-axiomatic`. Here are the helpers used by most of the tests in this section. Entries listed
 3081 as $f\langle N \rangle$ mean a family of functions $f1$, $f2$, $f3$ and so on, where N is typically the level.

- 3082 ▷ $pte\langle N \rangle(va)$: The (intermediate) physical address of the level N entry in the default translation
 3083 tables that maps va .
- 3084 ▷ $desc\langle N \rangle(va)$: The 64-bit descriptor from the initial state of the level N entry that maps va (the
 3085 value of $pte\langle N \rangle(va)$ in the initial state).
- 3086 ▷ $page(va)$: The page number that va is in (equivalently: $va \gg 12$).
- 3087 ▷ $mkdesc\langle N \rangle(oa=pa)$: A 64-bit descriptor for a valid leaf entry at level N where the output address is
 3088 given by the `oa` parameter.
- 3089 ▷ $mkdesc\langle N \rangle(table=pa)$: A 64-bit descriptor for a valid table entry at level N where the next-level-table
 3090 address is given by the `table` parameter.

¹The `ELR_ELx` (Exception-link-register) defines the return address of an exception to `ELx`. Translation faults, by default, return to the instruction that generated them.



This test is a variation on the standard write-read coherence test, CoWR, but where the VA is replaced with two distinct VAs, which both alias to the same PA.

The initial state is a configuration with two virtual addresses, x and y, which are both mapped to the physical address pa1, whose initial value is 0. The thread then stores 1 to x, then loads y. It is then forbidden for this load to read 0.

The Armv8-A architecture reference manual describes data caches as being physically-indexed [12, D5.11.1 (p4931)] and so accesses via the same PA are ‘fully coherent’. Further discussions with Arm clarify that this implies not just this coherence test, but that all prior data memory behaviours previously examined still apply when subjected to aliasing.

Figure 8.3: Test CoWR.alias

8.2 Aliased data memory

Much of the previous work on relaxed memory has been concerned with what we shall call ‘data memory’: the weak behaviour of concurrent loads and stores to memory, in the usermode fragments of the ISA. For Arm, we shall see that these previous models were implicitly assuming that all locations in the test were virtual addresses, with well-formed, constant, and injective, address translation mappings, which mapped all locations as readable, writable, and executable, normal cacheable memory.

Consider a non-injective mapping. Such mappings give rise to *aliasing*: the situation where two distinct virtual addresses in the same address space map to the same output physical address. This section will explore how the behaviours of those data memory tests change in the presence of aliasing.

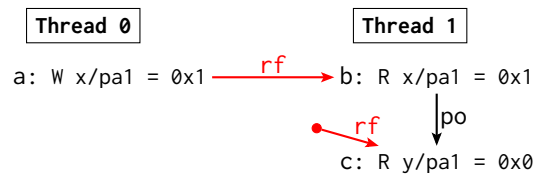
8.2.1 Virtual coherence

For data memory accesses, one of the most fundamental guarantee that architectures provide is *coherence*: in any execution, for each memory location, there is a total order of the accesses to that location, consistent with the program order of each thread, with reads reading from the most recent write in that order. Hardware implementations provide this, despite their elaborate cache hierarchies and out-of-order pipelines, by a combination of coherent cache protocols and pipeline hazard checking, identifying and restarting instructions when possible coherence violations are detected.

For Arm, coherence is with respect to physical addresses [12, B2.3.1 (p157)][12, D5.11.1 (p4931)]. This means that if two virtual addresses alias to the same physical address, then:

- ▷ A load from one virtual address cannot ignore a program-order previous store to the other, as seen in the CoWR.alias test (Figure 8.3).
- ▷ A load from one virtual address cannot ignore the write that a program-order previous load of the other address saw (CoRR0.alias+po (Figure 8.4, p.110), CoRR2.alias+po (Figure 8.5, p.110)).
- ▷ A load from one virtual address can have its value forwarded from a store to the other, and similarly on a speculative branch (MP.alias3+rfi-data+dmb (Figure 8.6, p.111), PPOCA.alias (Figure 8.6, p.111)).

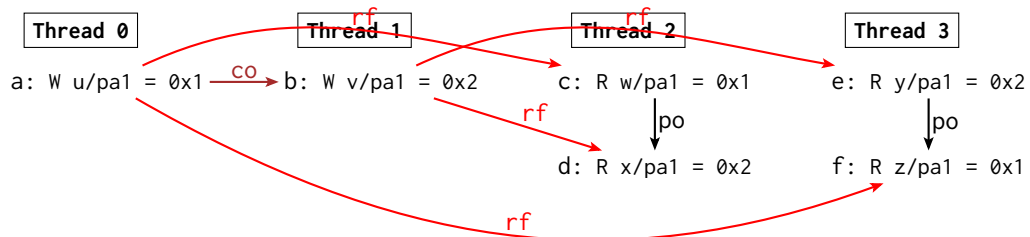
CoRR0.alias+po		AArch64
Initial state: $x \mapsto pa1, y \mapsto pa1, *pa1 = 0, 0:X0=1, 0:X1=x, 1:X1=x, 1:X3=y$		
Thread 0	Thread 1	
STR X0, [X1]	LDR X0, [X1]	LDR X2, [X3]
Forbidden: $1:X0=1 \ \& \ 1:X2=0$		



This test is a variation of the data memory CoRR0 test, where one of the loads has been replaced with a load of a distinct virtual address which aliases to the same underlying physical address. Note that, like the original test, it is forbidden to read from the initial state in the later load, as this would violate coherence: exactly what the earlier text from the manual explicitly forbade.

Figure 8.4: Test CoRR0.alias+po

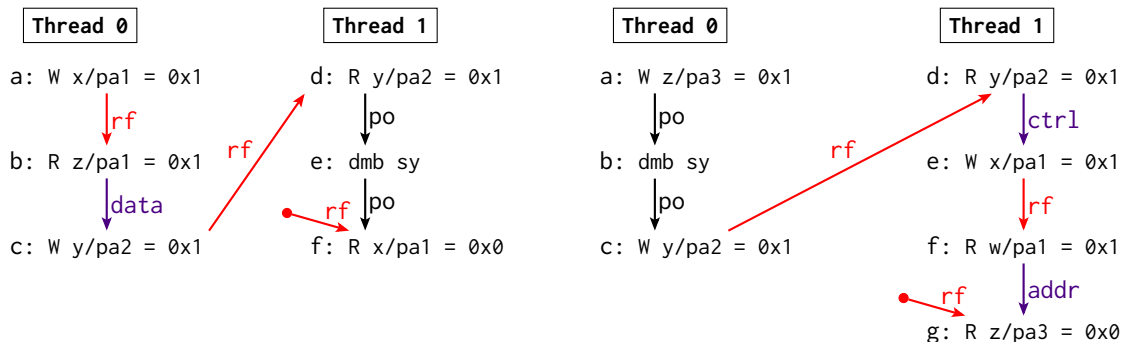
CoRR2.alias+po				AArch64
Initial state: $u \mapsto pa1, v \mapsto pa1, w \mapsto pa1, x \mapsto pa1, y \mapsto pa1, z \mapsto pa1, *pa1 = 0, 0:X0=1, 0:X1=u, 1:X0=2, 1:X1=v, 2:X1=w, 2:X3=x, 3:X1=y, 3:X3=z$				
Thread 0	Thread 1	Thread 2	Thread 3	
STR X0, [X1]	STR X0, [X1]	LDR X0, [X1] LDR X2, [X3]	LDR X0, [X1]	LDR X2, [X3]
Forbidden: $2:X0=1 \ \& \ 2:X2=2 \ \& \ 3:X0=2 \ \& \ 3:X2=1$				



This test is a variation of the data memory CoRR2 test. Here, there are many options for considering aliasing; we present here the maximally aliased version where each individual store and load uses a distinct virtual address, but where all those virtual addresses alias to the same physical one. This gives us a classic coherence shape, where it is forbidden for different threads to observe writes to the same physical location in different orders.

Figure 8.5: Test CoRR2.alias+po

MP.alias3+rfi-data+dmb	AArch64	PPOCA.alias	AArch64
Initial state: x -> pa1, y -> pa2, z -> pa1, *pa1 = 0, *pa2 = 0, 0:X0=1, 0:X1=x, 0:X3=z, 0:X5=y, 1:X1=y, 1:X3=x		Initial state: w -> pa1, x -> pa1, y -> pa2, z -> pa3, *pa1 = 0, *pa2 = 0, *pa3 = 0, 0:X0=1, 0:X1=z, 0:X2=1, 0:X3=y, 1:X1=y, 1:X2=1, 1:X3=x, 1:X5=w, 1:X7=z	
Thread 0	Thread 1	Thread 0	Thread 1
STR X0, [X1] LDR X2, [X3] STR X2, [X5]	LDR X0, [X1] DMB SY LDR X2, [X3]	STR X0, [X1] DMB SY STR X2, [X3]	LDR X0, [X1] CBNZ X0, LO LO: STR X2, [X3] LDR X4, [X5] EOR X8, X4, X4 LDR X6, [X7, X8]
Allowed: 1:X0 = 1 & 1:X2 = 0		Allowed: 1:X0 = 1 & 1:X4 = 1 & 1:X6 = 0	



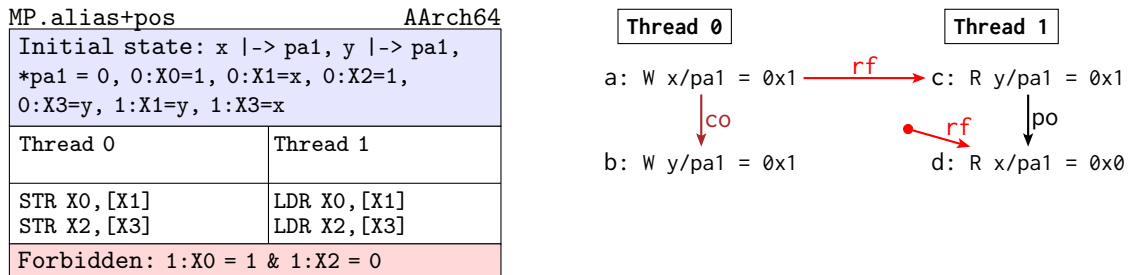
These tests are variations of the standard PPOCA and MP+rfi-data+dmb tests, but with some aliasing. Both are examples of *forwarding*: thread-locally reading from a write before it has been propagated to memory. These two tests, determined to be allowed architecturally from our discussions with Arm, show that the processor can forward from a write even if the read was for a different virtual address so long as the physical addresses match, even down a speculative path.

Figure 8.6: PPOCA.alias and MP.alias3+rfi-data+dmb: forwarding tests with aliasing.

3116 8.2.2 Aliasing different locations

3117 In the previous section, we explored taking tests over a single location, and rewriting the test to use many
 3118 locations, which all alias to the same address. One can also take a test that has multiple locations and
 3119 make some of them alias to the same address.

3120 Multi-location data memory tests, which are architecturally allowed, may become forbidden in the presence
 3121 of aliasing. For example, starting from the traditional MP+pos test, aliasing the two locations to the
 3122 same physical address gives the forbidden MP.alias+pos test (Figure 8.7). This new test is, essentially,
 3123 equivalent to the old CoRR0 test: coherence with two writes and two reads to the same location.



Because x and y alias to the same physical address pa1, the two loads (c and d) read the same location, and so cannot read different writes out-of-order.

Figure 8.7: Test MP.alias+pos

3124 8.2.3 Might be same (physical) address

3125 There is a corner case that we now should consider. For load and store instructions, when the last register
 3126 used in the calculation of the address is read, the address becomes known. In Flat, this may permit some
 3127 program-order-later instructions to know they operate over disjoint footprints and begin execution early.

3128 With the introduction of address translation, however, the point where instructions know the footprints of
 3129 previous instructions happens much later, after the whole translation table walk is performed. Between
 3130 the read of the register and the completion of the translation table walk, other instructions may perform
 3131 some part of their functionality. This may include reading from a different virtual address, before the
 3132 physical address of a program-order-previous instruction is known, but after the virtual address is known.

3133 8.2.4 Must not be same physical address

3134 One might expect that, when deciding whether to propagate a store, if the offset within the page of virtual
 3135 memory is distinct to that of the in-flight program-order earlier instructions, then the write could go ahead
 3136 early, knowing that the access could not be to the same physical address as any of those instructions. This
 3137 is, perhaps surprisingly, not the case: although the accesses definitely will not access the same physical
 3138 address, the program-order earlier access may still fault, meaning the write will not be reached. This
 3139 means that writes must wait for program-order-earlier translations to finish (or at least, be known to not
 3140 fault) before they can be propagated to other threads.

3141 8.3 What can be cached in TLBs

3142 As was described in §7.7, Arm hardware can have TLBs, caching previously seen translations. This
 3143 caching is, however, restricted, both in what information a TLB must cache when it does so, but also in
 3144 what kind of information it is not permitted to cache at all.

3145 We explore this by first examining the structure of a concrete implementation's MMU, before abstracting
 3146 away details to produce an abstract model, which we then use to explain the behaviours throughout the
 3147 rest of the chapter.

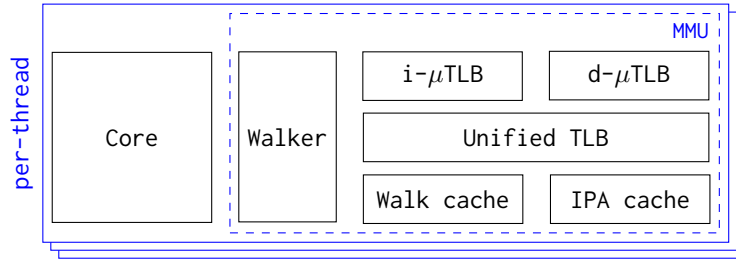


Figure 8.8: Block diagram of the Arm Cortex-A53 memory-management-unit [73].

8.3.1 Microarchitectural TLBs

Here we make a clear distinction between the actual *microarchitectural* translation caching one may encounter inspecting hardware, and the architectural model being discussed here.

While there are possibly many different ways to describe the same architectural intent, we carefully choose one which will make building tooling, extending the model, discussions with architects, and explaining individual tests easier. We will first look at a specific example to pin down terminology and gain some intuition for hardware, before giving a model MMU and TLB that abstracts away from the details.

Microarchitectural MMU – A53 Let us explore more closely how the actual hardware fill and walk works on a modern microprocessor. The Arm Cortex A53 is an Arm-designed application class processor. Previous relaxed memory work included exercising this core design extensively during litmus testing validation of the models, finding it to be relaxed, exhibiting many relaxed behaviours, but not aggressively so. This makes the A53 a good candidate as a demonstrator of an average relaxed processor design. While other processors by Arm are more aggressive in their optimisations, the MMU and TLB layout of the A53 seems typical: other cores generally have comparable TLB configurations [92, 93, 94, 95, 96].

The Arm A53 Technical Reference Manual (TRM) describes, in detail, the structure of the memory management unit [73, 5-2] of the A53, and its constituent parts. Figure 8.8 contains a block diagram representing the key structures in the A53’s memory management unit.

Each core has its own MMU, and each MMU contains:

- ▷ the walker, which actually does the translation table walk;
- ▷ one instruction micro-TLB (denoted $i\text{-}\mu\text{TLB}$ in the diagram);
- ▷ one data micro-TLB ($d\text{-}\mu\text{TLB}$);
- ▷ one unified TLB;
- ▷ one walk cache; and,
- ▷ one IPA cache.

The microarchitectural TLBs store translations: virtual to physical mappings, plus permissions and so-on, tagged with their translation context: the translation regime, ASID, VMID, virtual or intermediate-physical address, and so on. The TLBs are arranged hierarchically, with small, 10-entry, micro TLBs for instruction and data streams separately, and one large 512-entry unified TLB. On a TLB miss, the MMU performs a translation table walk using the walker.

When it begins this walk, the MMU first checks the walk cache. Walk caches store mappings from virtual address to the physical address of the last level translation table – commonly seen in modern MMUs. When the walk cache has an entry, the walker can skip over most of the walk and directly read the leaf entry out the last-level table.

If a second stage of translation is required during the walk, the IPA cache is used (and may be used many times during the same walk). The IPA cache stores mappings from intermediate physical to physical memory — without an associated virtual address — which can be used during both the final stage 2 walk, and any intermediate stage 2 walks during a stage 1 walk.

The MMU is free to save the result of any translation table walk into these structures, including for walks due to speculation, prefetching, or architectural execution. This allows the MMU to perform a walk for any arbitrary VA or IPA, at any point in time.

8.3.2 Model MMU

To abstract away from any specific microarchitecture, we imagine modelling the MMU as if it were a separate asynchronous unit, one for each thread, each with its own TLB.

Later, we will see tests that justify and ground this particular choice of abstraction, and we will explore the consequences of this model in more rigorous detail. For now, we can imagine this model MMU as a set of (concurrently) executing translation table walks, and a cache of translation table entries.

Model TLB entries In general, the architecture permits hardware to cache whatever information from the translation process the hardware sees fit. This may include the output of whole translation table walks (complete virtual to physical mappings) or individual translation table entries, or even the result of partial walks (the address of the last-level table, for example).

It would not be feasible to enumerate all the possible shapes of TLBs, and the kinds of information they can cache. Instead, we define a *model TLB*. This model TLB acts as a cache of writes of translation table entries, each tagged with some context. This allows the model to cache any combination of valid entries in a translation table walk: weak enough to allow all currently known TLB implementations, but strong enough to not break any of the guarantees software requires. These guarantees are explored, in detail, in §8.4 and §8.5.

Each entry in the model TLB contains the information about the write itself: the physical address of the entry, and the cached 64-bit entry. Each entry must also be tagged with some contextual information. This contextual information is required for two distinct purposes, and so we split into two parts: an architectural context which defines the scope of a cached translation table entry, that is, for which translations the cached value is permitted to be used; and an extended model context, which serve as tags for TLB maintenance operations to target.

This contextual information is:

- ▷ the architectural context, used for matching entries in the TLB:
 - the translation regime;
 - the VMID;
 - the ASID (or a ‘global indicator’);
 - and the virtual address, intermediate physical address, and/or physical address of the translation.
- ▷ the model context, used for targeted TLB maintenance:
 - the translation stage and level at which the write was used;
 - the system register values used in the translation (those which can be cached);
 - and, for an entry used for a Stage 1 translation, whether it has been invalidated at both stages.

Operationally, one can imagine performing a translation using the model MMU by doing a full translation table walk, but being able to optionally satisfy any read during that walk from a matching entry in the model TLB which matches the architectural context and input address. Any behaviour exhibited by a specific micro-architectural MMU and TLB configuration, and therefore all the litmus tests in this chapter, should be consistent with this model.

TLB fills Hardware has a variety of mechanisms which may lead to a translation table walk: direct architectural execution of instructions, pre-fetching of data or instructions, and speculation down branches. These translation table walks may result in TLB misses, and those misses then result in reads from memory and the MMU ‘filling’ the TLB with a copy of the information it can use in future.

Arm does not enumerate all the possible speculation machinery or prefetchers, so instead we opt for a model that defines a wide envelope of behaviour, weak enough to permit a range of plausible hardware implementations: at any point in time, any thread’s MMU can spontaneously perform a translation table walk for any virtual or intermediate-physical address for the current architectural context (VMID, ASID, etc, as in ‘[Model TLB entries](#)’), and any reads that the translation table walk performs can either read from other TLB entries, or perform a non-TLB read of memory and then potentially cache a copy of the write it reads from in the TLB, tagged with the context information from the walk. The behaviour of those non-TLB reads is explored more in §8.4.

8.3.3 Invalid entries

It is architecturally forbidden to cache information from attempted translations which result in translation faults, access flag faults, or address size faults (note that a translation table walk may give rise to other faults as well, as discussed in §7.4, such as permission faults and alignment faults, which do not impose restrictions on TLB caching). More specifically, a TLB entry cannot be a write of a translation table entry which is the direct cause of such a fault. In particular, the TLB cannot cache translation table entries whose valid bit is not set.

This is important, as it gives software a mechanism in which it can safely write a new mapping without potentially having multiple entries in the TLB for the same virtual address, as can be seen in the tests in §8.4, which forms a key part of the software pattern for Arm to update a previously-valid pagetable entry: *break-before-make*, discussed in §8.6.5.

8.4 Reads not from TLB

The requirement that invalid entries are not cached in the TLB gives us a way to directly observe the behaviour of non-TLB reads: translation table reads which directly result in a translation fault *must not* have come from a TLB read. We will see that these reads have some important properties that software can rely on, but that some of those properties will depend on certain architecture features being enabled (namely ‘Enhanced Translation Synchronisation’, FEAT_ETS, see §8.4.3).

In this section we will explore the properties these reads have, and the guarantees software can rely on. We will see that these reads are affected by thread-local re-ordering, to an even greater extent than data memory reads, and explore the synchronization that recovers the sequential semantics. We will further see how these reads from the translation table walk relate to data memory reads, with respect to coherence, multi-copy atomicity, write forwarding and so on. Finally, we will see how the FEAT_ETS architectural feature can change the required synchronization software needs to perform.

8.4.1 Out-of-order execution

First, let us consider whether reads that do not come from the TLB (non-TLB reads) preserve the original program order. One of the simplest questions one can ask is whether a translation-table-walk non-TLB read can ignore a program-order previous store. This scenario is captured by the `CoWTf.inv+po` test (Figure 8.9, p.116). Starting with a VA (`x`) initially invalid at level 3, which therefore cannot have its level 3 entry cached in any TLB (directly or indirectly), the test overwrites the invalid entry with a new valid entry pointing to the physical address `pa1`. Program-order later, the thread then attempts to read `x`. The question is whether the read of `x` can read-from the old translation table entry, generating a translation fault. This result is architecturally allowed, and readily observed on hardware.

One explanation that suffices to allow this outcome is that the instructions can be locally re-ordered; the translation table walk of the later load instruction can happen much earlier than the program-order previous store, and satisfy its read from memory first. Similarly, the reads of a translation table walk can be locally re-ordered with respect to program-order earlier loads of the translation table entry, as demonstrated in the `CoRpteTf.inv+po` test (Figure 8.10, p.116).

A translation table walk read may not, in general, be re-ordered past program-order-later stores. This is consistent with the description in §8.2.3, as the program-order later store might not architecturally happen if the translation table walk read were to fault. So, the later writes are speculative until the translation has finished, preventing the write from propagating until then. This forbids both the re-ordering of the propagation of the write to other threads (`LB.TT.inv+pos` (Figure 8.11, p.117)) with program-order earlier translation table walks, and translations reading from program-order later writes (`CoTW1.inv` (Figure 8.12, p.117)).

CoWtF.inv+po		AArch64
Initial state: x -> invalid, y -> pa1, *pa1 = 1, 0:X0=desc3(y), 0:X1=pte3(x), 0:X3=x		
Thread 0	Thread0 El1 Handler	
STR X0, [X1] LDR X2, [X3]	MOV X2, #0 MRS X20, ELR_EL1 ADD X20, X20, #4 MSR ELR_EL1, X20 ERET	
Allowed: 0:X2 = 0		

Thread local re-ordering lets the translation (b1) of the load instruction happen earlier than the write to the translation table (a). This allows the load to trigger a data abort (a translation fault, b2).

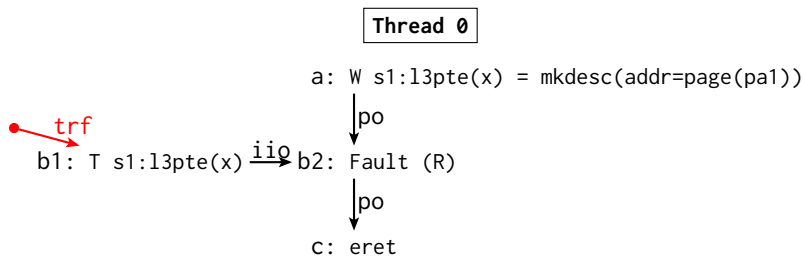


Figure 8.9: Test CoWtF.inv+po

CoRpteTf.inv+po		AArch64
Initial state: x -> invalid, y -> pa1, *pa1 = 1, 0:X0=desc3(y), 0:X1=pte3(x), 1:X1=pte3(x), 1:X3=x		
Thread 0	Thread 1	Thread1 El1 Handler
STR X0, [X1]	LDR X0, [X1] LDR X2, [X3]	MOV X2, #0 MRS X13, ELR_EL1 ADD X13, X13, #4 MSR ELR_EL1, X13 ERET
Allowed: 1:X0 = desc3(y) & 1:X2=0		

The translation read (event c1) can be re-ordered with respect to the program-order previous load of l3pte(x) (b), even though the load read the new translation table entry, for the same location the translation reads from.

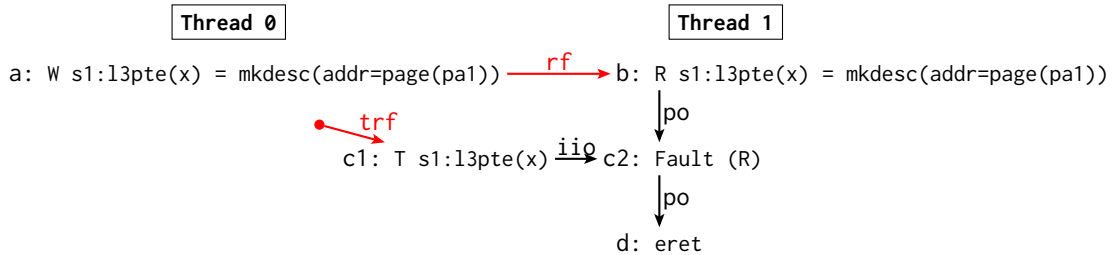
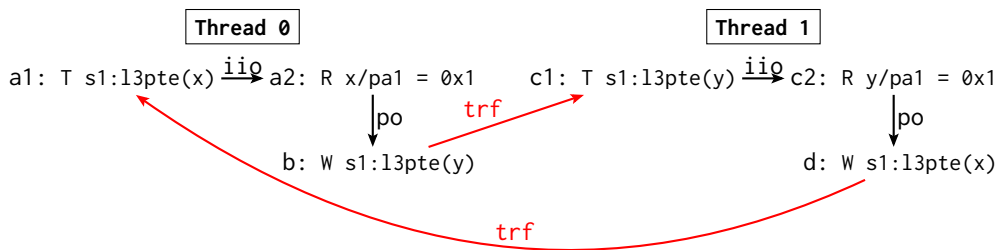


Figure 8.10: Test CoRpteTf.inv+po

LB.TT.inv+pos		AArch64	
Initial state: x -> invalid, y -> invalid, *pa1 = 1, 0:X1=x, 0:X2=mkdesc3(oa=pa1), 0:X3=pte3(y), 1:X1=y, 1:X2=mkdesc3(oa=pa1), 1:X3=pte3(x)			
Thread 0	Thread 1	Thread0 E11 Handler	Thread1 E11 Handler
MOV X0,#0 LDR X0,[X1] STR X2,[X3]	MOV X0,#0 LDR X0,[X1] STR X2,[X3]	MRS X13,ELR_EL1 ADD X13,X13,#4 MSR ELR_EL1,X13 ERET	MRS X13,ELR_EL1 ADD X13,X13,#4 MSR ELR_EL1,X13 ERET
Forbidden: 0:X0 = 1 & 1:X0=1			



The writes to the translation tables (b and d) are forbidden from propagating to other threads before the program-order earlier translations (a1 and c1) are satisfied.

Figure 8.11: Test LB.TT.inv+pos

CoTW1.inv		AArch64	
Initial state: x -> invalid, y -> pa1, *pa1 = 1, 0:X1=x, 0:X2=desc3(y), 0:X3=pte3(x)			
Thread 0	Thread0 E11 Handler		
LDR X0,[X1] STR X2,[X3]	MOV X0,#0 MRS X13,ELR_EL1 ADD X13,X13,#4 MSR ELR_EL1,X13 ERET		
Forbidden: 0:X0 = 1			

The store to the translation table (b) cannot be re-ordered with the program-order earlier translation table walk (a1), preventing that walk from reading from the store.

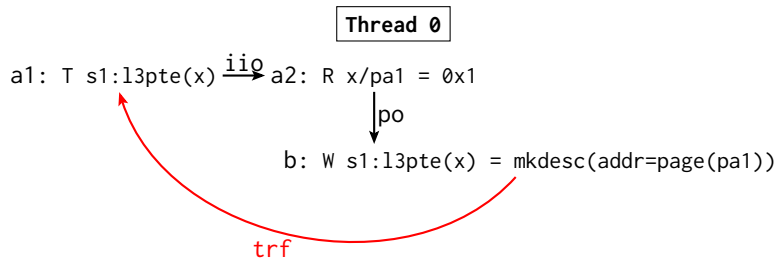


Figure 8.12: Test CoTW1.inv

CoWtF.inv+dsb-isb		AArch64
Initial state: x -> invalid, y -> pa1, *pa1 = 1, 0:X0=desc3(y), 0:X1=pte3(x), 0:X3=x		
Thread 0	Thread0 El1 Handler	
STR X0, [X1] DSB SY ISB LDR X2, [X3]	MOV X2, #0 MRS X20, ELR_EL1 ADD X20, X20, #4 MSR ELR_EL1, X20 ERET	
Forbidden: 0:X2 = 0		

The write to the translation table (a) is ordered before the non-TLB read of the entry (d1) because of the intervening DSB; ISB sequence, creating local order. This ordering ensures that the non-TLB read respects the coherence order up to the point of the write a, preventing the non-TLB read from reading from a write coherence-before a.

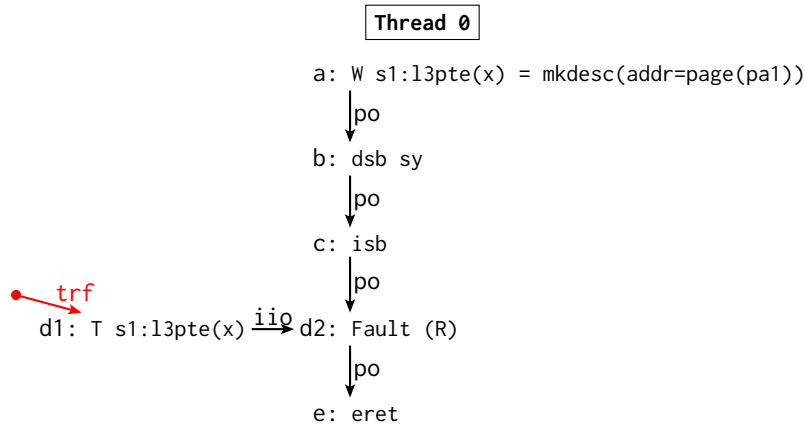


Figure 8.13: Test CoWtF.inv+dsb-isb

8.4.2 Enforcing thread-local ordering

3281

3282 Since non-TLB reads do not necessarily preserve the program order, it appears that there are no coherence
 3283 guarantees one can make about them. However, by introducing some thread-local ordering constructs, we
 3284 can recover some of the strong guarantees we are used to, e.g. to ensure that program-order later accesses
 3285 use the new translations.

3286 To force a non-TLB read to happen after some program-order earlier event, we can insert the two-
 3287 instruction sequence DSB SY ; ISB between them. The DSB (‘Data Synchronization Barrier’) waits for all
 3288 loads to satisfy and for all stores to have finished and be visible to translation table walkers, then the
 3289 ISB (‘Instruction Synchronization Barrier’) ensures that program-order later instructions begin execution
 3290 (including translation table walks) after the completion of the DSB, and therefore the earlier stores.

3291 **Locally-ordered-previous writes** If we introduce this sequence into the previous `CoWtF.inv+po` test
 3292 (Figure 8.9, p.116), we obtain the `CoWtF.inv+dsb-isb` test (Figure 8.13), which is forbidden by Arm.
 3293 This is because the non-TLB reads, in the absence of non-coherent TLB caching structures (discussed
 3294 more in §8.6.1), will read from the coherent storage subsystem, and so will be required to see the new
 3295 write, or something coherence-after it.

3296 **Locally-ordered-previous reads** If a program-order-previous load has already seen some other-thread
 3297 write, either through a translation (`CoTtF.inv+dsb-isb` (Figure 8.14, p.119)), or through a normal data
 3298 load of the translation table (`CoRpteTf.inv+dsb-isb` (Figure 8.15, p.119)), then translation table non-TLB
 3299 reads which are ordered after that read must also see that write, or a write coherence-after it. These tests
 3300 use the DSB; ISB sequence previously described, but any ordering to the translation table walk (described
 3301 in §8.4.3) suffices.

3302 Microarchitecturally, this is because translation table walkers behave as separate ‘observers’ [68, p.14726].
 3303 Essentially, the MMU is a sibling element to the processor, accessing memory through the same coherent
 3304 mechanism as the primary CPU. This ‘separate observers’ principle may seem a reasonable model at first,
 3305 however, we will see later on in §8.4.4 where it breaks down as an architectural model.

CoTTf.inv+dsb-isb		AArch64
Initial state: x -> invalid, y -> pa1, *pa1 = 1, 0:X0=desc3(y), 0:X1=pte3(x), 1:X1=x, 1:X3=x		
Thread 0	Thread 1	Thread1 EL1 Handler
STR X0, [X1]	LDR X2, [X1] MOV X0, X2 DSB SY ISB LDR X2, [X3]	MOV X2, #0 MRS X13, ELR_EL1 ADD X13, X13, #4 MSR ELR_EL1, X13 ERET
Forbidden: 1:X0 = 1 & 1:X2=0		

The second translation-table non-TLB read of x (e1) is locally ordered after the first translation table walk (b1) because of the intervening dsb; isb sequence, and so cannot see a write coherence before the write the earlier (b1) translation-read read from.

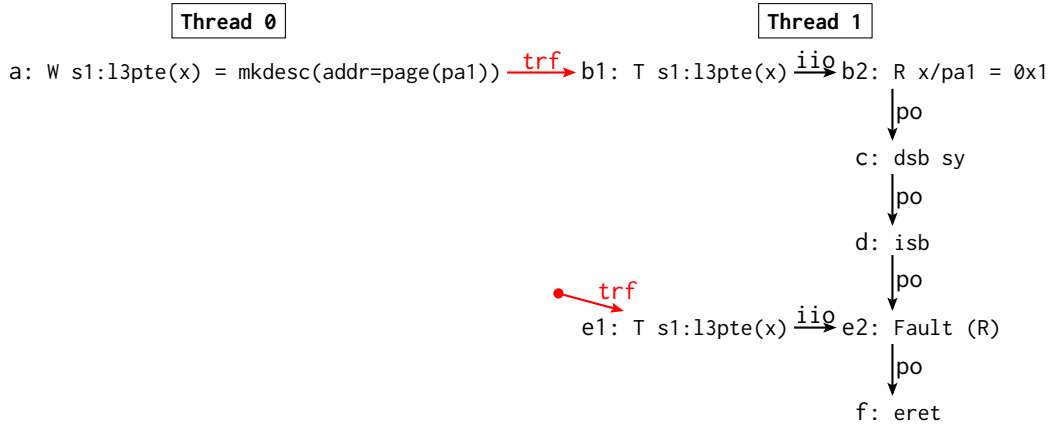


Figure 8.14: Test CoTTf.inv+dsb-isb

CoRpteTf.inv+dsb-isb		AArch64
Initial state: x -> invalid, y -> pa1, *pa1 = 1, 0:X0=desc3(y), 0:X1=pte3(x), 1:X1=pte3(x), 1:X3=x		
Thread 0	Thread 1	Thread1 EL1 Handler
STR X0, [X1]	LDR X0, [X1] DSB SY ISB LDR X2, [X3]	MOV X2, #0 MRS X13, ELR_EL1 ADD X13, X13, #4 MSR ELR_EL1, X13 ERET
Forbidden: 1:X0 = desc3(y) & 1:X2=0		

The final translation table walk of x (e1) cannot be re-ordered with the program-order previous load of pte3(x) (b), because of the intervening DSB; ISB sequence. The non-TLB translation read of pte3(x) (e1) therefore must read from the same write as the earlier load, or something coherence-after it.

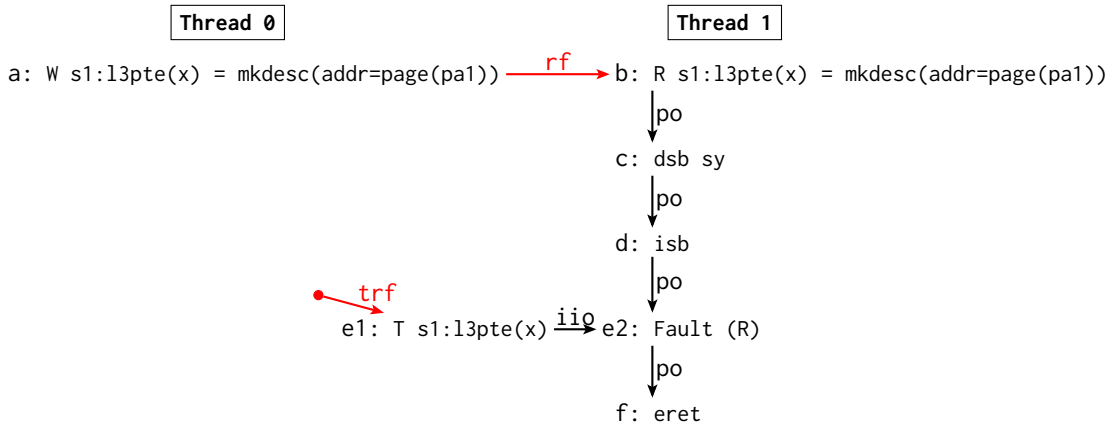


Figure 8.15: Test CoRpteTf.inv+dsb-isb

CoTTf.inv+ctrl-isb		AArch64
Initial state: x -> invalid, y -> pa1, *pa1 = 1, 0:X0=desc3(y), 0:X1=pte3(x), 1:X1=x, 1:X3=x		
Thread 0	Thread 1	Thread1 EL1 Handler
STR X0, [X1]	MOV X0, #0 LDR X0, [X1] EOR X4, X0, X0 CBNZ X4, LC00 LC00: ISB MOV X2, #0 LDR X2, [X3]	MRS X13, ELR_EL1 ADD X13, X13, #4 MSR ELR_EL1, X13 ERET
Forbidden: 1:X0 = 1 & 1:X2=0		

Control-ISB locally-orders the later translation table walk (d1) after the resolution of the control flow, which happens only after the satisfaction of the read b2.

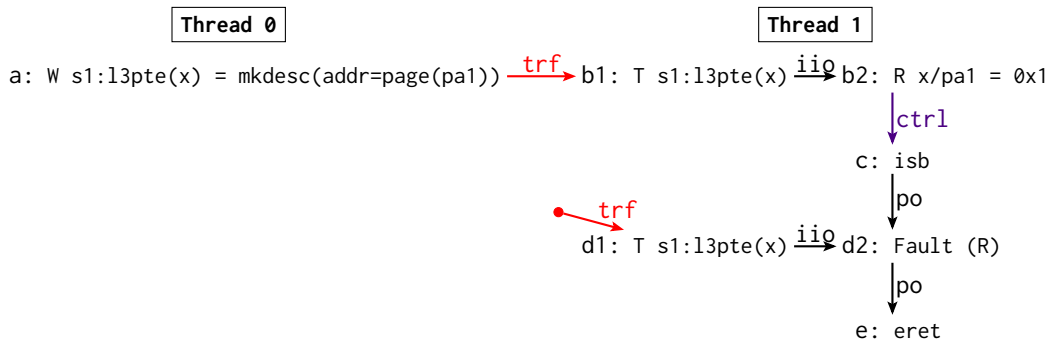


Figure 8.16: Test CoTTf.inv+ctrl-isb

3306 **Instruction synchronisation barrier and control dependencies** The ISB instruction naturally orders all
 3307 translation table walks of program-order later instructions with the ISB itself. This is because the ISB
 3308 effectively restarts all program-order later instructions, including any translations they do.

3309 However, an ISB is not naturally ordered with respect to program-order *earlier* instructions. That is why
 3310 we introduced a DSB in the previous tests, without which they would be allowed. A control-dependency to
 3311 the ISB would also work (CoTTf.inv+ctrl-isb (Figure 8.16)).

3312 **Address dependencies** In previous work, address dependencies were assumed fundamental. Now we can
 3313 define what an address dependency is: dataflow into the translation table walk. Address dependencies
 3314 remain a strong way to order events. Arm does not permit observable speculation of the values or addresses
 3315 of explicit reads and writes to memory. This means that a translation table walk will not start until
 3316 after its address dataflow-dependent registers are fully determined. Note, that this does not mean that
 3317 pre-fetching and caching of the walk cannot happen: it's just that the architectural translation table walk
 3318 must retrieve any cached values after it is known what the address will be. Therefore, non-TLB translation
 3319 reads are locally-ordered-after any read whose value flows into that non-TLB read, as demonstrated in
 3320 CoRpteTf.inv+addr (Figure 8.17, p.121).

3321 **Memory barriers** Much of the earlier work in relaxed-memory concurrency was dedicated to the behaviour
 3322 of *barriers*. The Arm data memory barrier (DMB) creates ordering between memory events program-order
 3323 earlier than the barrier, and memory events program-order after the barrier.

3324 We will see that this applies to *explicit* memory events only: the principal reads and writes that load and
 3325 store instructions perform, not the implicit reads and writes they do during translations (or instruction
 3326 fetching, see Part I).

3327 Ordering of the explicit memory events does not, automatically, induce ordering between those explicit
 3328 events and any reads due to translation table walks performed by those instructions. In the next subsection,
 3329 we will see how FEAT_ETS (§8.4.3) extends the architecture to include more orderings between translations
 3330 and other memory events in the same thread.

CoRpteTf.inv+addr		AArch64
Initial state: x -> invalid, y -> pa1, *pa1 = 1, 0:X0=desc3(y), 0:X1=pte3(x), 1:X1=pte3(x), 1:X3=x		
Thread 0	Thread 1	Thread1 EL1 Handler
STR X0, [X1]	LDR X0, [X1] EOR X4, X0, X0 LDR X2, [X3, X4]	MOV X2, #0 MRS X13, ELR_EL1 ADD X13, X13, #4 MSR ELR_EL1, X13 ERET
Forbidden: 1:X0 = desc3(y) & 1:X2=0		

The address dependency from the load b to the second load, orders the reads due to the translation table walk of that load (in particular, c1) after b. Since c1 is a non-TLB read, it cannot read from a write coherence-before the write b read from.

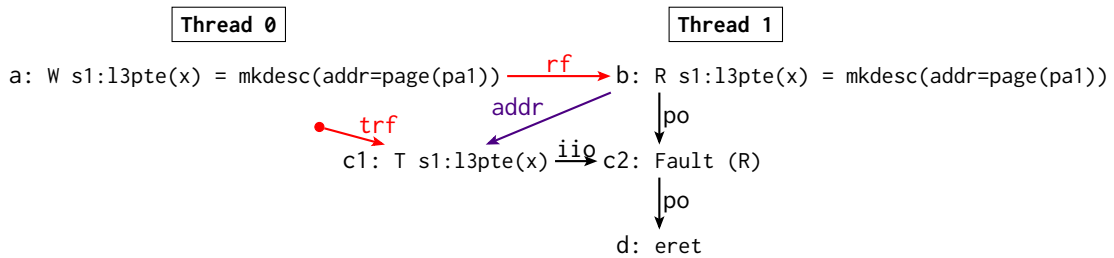


Figure 8.17: Test CoRpteTf.inv+addr

3331 Figure 8.18 shows a simple coherence test, with a data memory barrier between a store to the translation
 3332 tables and a load whose translation table walk might read from that. We see that the DMB does not enforce
 3333 that the translation table walk sees the update to the translation tables. From the previous tests, we
 3334 know this means that the translation table walk happened (microarchitecturally) before the store was
 3335 propagated to memory.

3336 **The Arm DMB vs DSB instructions** Arm provides two memory barrier instructions: DMB ('data memory
 3337 barrier') and DSB ('data synchronisation barrier'). The base intent is that DMB orders explicit memory
 3338 accesses, whereas DSB is a strictly stronger barrier also ordering some implicit accesses, and other barriers
 3339 and cache maintenance (including TLB invalidation). This means that, for any litmus test with a DMB,
 3340 a DSB of the same access kind could be substituted, and the resulting test is no weaker. Over time, the
 3341 architectural intent around how barriers order implicit events has changed, and is still subject to change.

CoWtF.inv+dmb		AArch64
Initial state: x -> invalid, y -> pa1, *pa1 = 1, 0:X0=desc3(y), 0:X1=pte3(x), 0:X3=x		
Thread 0	Thread0 El1 Handler	
STR X0, [X1] DMB SY LDR X2, [X3]	MOV X2, #0 MRS X20, ELR_EL1 ADD X20, X20, #4 MSR ELR_EL1, X20 ERET	
Allowed iff not ETS 0:X2 = 0		

The non-TLB read c1 is not locally ordered after the write a, despite the intervening dmb sy barrier (b).

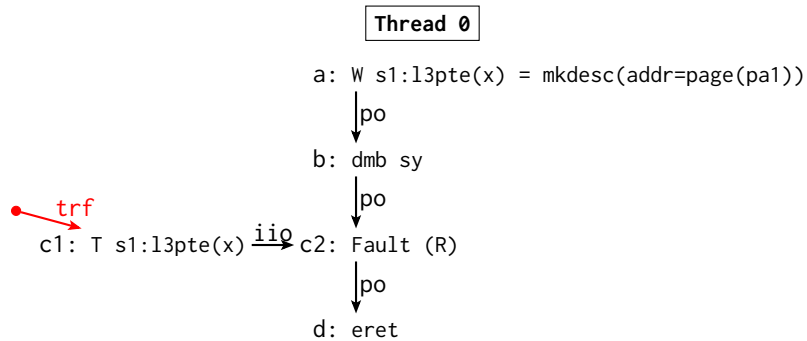


Figure 8.18: Test CoWtF.inv+dmb

8.4.3 Enhanced Translation Synchronization

Note: since this work there have been further extensions including FEAT_ETS2 and FEAT_ETS3. See the discussion at the end of the section for more details.

Recent versions of the Arm architecture require support for FEAT_ETS: Enhanced Translation Synchronization. This feature does not change the ISA directly, but instead requires implementations to enforce extra ordering.

When an instruction takes a translation fault, it is permitted (without ETS) to commit to taking that fault immediately. However, there may have been some thread-local dependencies or ordering into that instruction with the intent to order any explicit memory access (which do not happen if the instruction faults). The early committal to the fault effectively breaks such dependencies, allowing the handler to be executed before the dependency would have been resolved. This means, for example, if software were to create new translation tables and publish them, ‘later’ translations may have already committed to taking a fault, which software would observe as a *spurious* fault. This is what ETS tries to resolve.

The Arm Architecture Reference Manual says the following [12, D5.2.5 (p4802)]:

If FEAT_ETS is implemented, and a memory access RW1 is Ordered-before a second memory access RW2, then RW1 is also Ordered-before any translation table walk generated by RW2 that generates any of the following:

- ▷ A Translation fault.
- ▷ An Address size fault.
- ▷ An Access flag fault.

This prose description is phrased confusingly with reference to a seemingly non-existent event RW2, so requires some clarification: the scenario being described here is a case with two instructions, I_1 and I_2 , each either a load or store. Imagine I_1 and I_2 both executing to completion, without generating any translation, address size, or access flag faults. Then, each instruction would have generated one or more explicit memory events. For example, a store might generate up to 8 separate write events (one for each byte). Call those events E_{ij} for the j th explicit event of instruction I_i .

Each explicit event E_{ij} would have required a translation table walk, generating translation read events which we can call T_{ijk} for the k th translation-table-walk read for the j th explicit memory event for instruction I_i .

Then, if in some execution: I_2 generates a translation fault, address size fault, or access flag fault; E_{1n} would have been locally-ordered-before $E_{2,m}$ in an execution without the fault; and FEAT_ETS is enabled; then, E_{1n} is locally-ordered-before any translation table read $T_{2,m,-}$ in the execution with the fault. That is, we can imagine that in the execution with the fault there is a ‘ghost’ event corresponding to the fault in the place the explicit event *would have occurred* with the same ordering into it. This scenario is illustrated in Figure 8.19.

The intuition here is that, microarchitecturally, on implementations that support FEAT_ETS, when an instruction takes an exception, the access that caused it is re-tried once the prefix of instructions is non-restartable. This reduces spurious aborts by ensuring faults cannot come from an out-of-order read of (what is now) a stale value from memory.

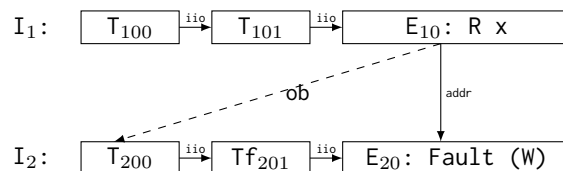


Figure 8.19: ETS Ghost events example: A load instruction (I_1) followed followed (in program order) by a store instruction (I_2), which faults. The address dependency means that the read event E_{10} is syntactically ordered-before the (ghost) write event E_{20} , and so the read event is ordered before the reads of the translation table walk for I_2 read from the TLB or memory (represented by the dashed *ob* line).

3375 **Other effects** The architecturally desired effect of FEAT_ETS seems to be that no additional context-
3376 synchronisation should be required to prevent spurious aborts, and that simple local orderings (barriers,
3377 dependencies) should be enough. To make this so, ETS must implicitly enforce more than just the
3378 aforementioned ordering constraints.

3379 Specifically, TLBI instructions must have stronger thread-local orderings to translation-table walks (de-
3380 scribed in more detail later); translation table walks must be (other) multi-copy atomic; and, translation
3381 table walk reads must be coherent and single-copy atomic.

3382 **non-ETS fragment** There is a question here as to whether we should consider the non-ETS behaviours
3383 of the architecture. On the one hand, hardware in use today is from a pre-ETS version of the architecture
3384 and so we cannot assume that the behaviour of those devices are consistent with ETS. On the other hand,
3385 ETS is a feature that is widely assumed by software, even if not present on hardware.

3386 In particular, Linux assumes implementations are ETS compatible even when they are not. Building
3387 models that capture the full extent of the non-ETS fragment would have questionable benefits as one
3388 would have to assume an ETS model when verifying software. Additionally, as ETS is becoming a
3389 mandatory feature, the concerns over non-ETS hardware will diminish over time. Finally, the semantics
3390 of this non-ETS fragment is still unclear; there are numerous questions, especially around forwarding and
3391 multi-copy atomicity generally, which are grey areas in the non-ETS fragment which Arm have yet to
3392 explicitly decide one way or another.

3393 For these reasons we will assume FEAT_ETS is present and enabled, unless explicitly stated otherwise.

3394 **Ordering to the translation table walk** We can now define which constructs give rise to local ordering
3395 into a translation table walk. Address dependencies, and locally-ordered context-synchronisation (in
3396 particular, the DSB; ISB sequence) always give rise to ordering to the translation table walks. Control
3397 dependencies, on their own, never give rise to such ordering. If using FEAT_ETS, then a plain DSB orders
3398 translation table walks of program-order later instructions after it. Other barriers may give ordering to
3399 the translation table walker, if using FEAT_ETS and the translation results in a translation fault, and those
3400 barriers would have ordered the event that would have happened otherwise.

3401 **ETS2, ETS3, and the future of ETS** Since this work was done, Arm have further refined the architecture;
3402 there are now two further extensions to ETS: FEAT_ETS2, which extends the behaviour described above to
3403 include ordering from all instructions program-order before the fault [68, D8.2.6.1]; and FEAT_ETS3, which
3404 extends this to all MMU faults [97, D8.2.6.1].

3405 FEAT_ETS, as described in this document, no longer exists. The memory model since developed by Arm, as
3406 found in the Arm architecture reference manual, describes the latest state of ETS [97, B2.3]. FEAT_ETS3 is
3407 now mandatory from Armv9.5.

3408 **8.4.4 Forwarding to the translation table walker**

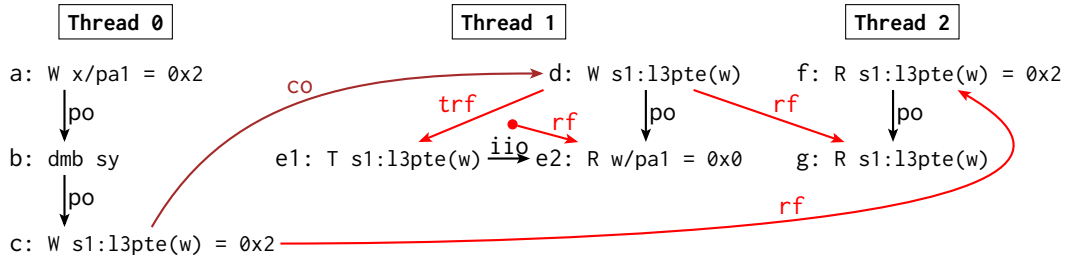
Note: our understanding of the architectural intent for forwarding to translation table walks has changed over time. See the discussion at the end of the section for more details.

3409
3410 Writes take time to propagate out to memory to other cores. One common performance optimization is
3411 *gathering*: collecting multiple writes together in a store buffer, to propagate them together. To maintain
3412 uniprocessor semantics, the core reads from its own store buffer, in effect, allowing it to read from writes
3413 before they've been propagated out to other cores. This behaviour is referred to as *write forwarding*.

3414 Although the translation table walker is described as a 'separate' observer, it is also part of the core that
3415 hosts it, and is allowed to read from that core's store buffer, effectively allowing writes to be 'forwarded'
3416 to the walker, as shown in the [R.TR.inv+dmb+trfi](#) test (Figure 8.20, p.125).

3417 **Changes to the architectural intent** Since this work was completed, the architectural intent has been
3418 further clarified: non-TLB reads, except with FEAT_nTLBPA, must be multicopy atomic with respect to
3419 data accesses.

R.TR.inv+dmb+trfi			AArch64
Initial state: w -> invalid, x -> pa1, *pa1 = 0, 0:X0=2, 0:X1=x, 0:X2=2, 0:X3=pte3(w), 1:X0=mkdesc3(oa=pa1), 1:X1=pte3(w), 1:X3=w, 2:X1=pte3(w)			
Thread 0	Thread 1	Thread 2	Thread1 EL1 Handler
STR X0, [X1] DMB SY STR X2, [X3]	STR X0, [X1] MOV X2, #1 LDR X2, [X3]	LDR X0, [X1] LDR X2, [X1]	MRS X13, ELR_EL1 ADD X13, X13, #4 MSR ELR_EL1, X13 ERET
Allowed: 1:X2=0 & 2:X0=2 & 2:X2=mkdesc3(oa=pa1)			



The write of the new valid entry (d) can be forwarded locally to the translation of w (e1) allowing the read of w (e2) to satisfy early. Thread 2 is an observer thread, witnessing that the write d happens after c.

Figure 8.20: Test R.TR.inv+dmb+trfi

8.4.5 Speculative execution

To facilitate out-of-order pipelines, the machine begins fetching and executing the next instruction before earlier instructions are finished. However, those earlier instructions may change the flow of execution through the program. Executing later instructions before those earlier instructions are finished means that those later instructions are being executed *speculatively*: the predicted control flow, or assumptions of independence between instructions, may turn out to be incorrect. When the control flow is mispredicted, or a speculative access leads a coherence violation, the speculated effects must be discarded.

When executing down a speculative path like this, there are additional restrictions that the core must adhere to. For example, stores should not be propagated out to memory, although they can still be read from by program-order-later reads in the same thread.

Since reads and writes can be performed speculatively, their associated translations must also be permitted to be performed speculatively. This is what permits tests such as [MP.RTf.inv+dmb+ctrl](#) (Figure 8.21, p.126) to see an old value for the translation table entry down a speculative path.

However, forwarding from a speculative write to the translation table walker is disallowed. Reads from normal memory have no side effects, but other locations, such as devices, are *read sensitive*: reads may effect the device state. Software protects such locations by marking them as device memory in the translation tables, or leaving them unmapped altogether. A still-speculative write could update the translation tables arbitrarily, including allowing reads to read-sensitive locations, so it must be forbidden for a translation read to read from still-speculative writes. The [MP.RT.inv+dmb+ctrl-trfi](#) test (Figure 8.22, p.126) demonstrates this, requiring that the translation table walk on the speculative path cannot read from the still-speculative store to the translation tables.

Instruction restarts A related, but separate, concept is that of instruction restarting. In the usermode memory model a read might be satisfied early, out-of-order with respect to program-order previous instructions, even before those instructions' accesses addresses are known. If such an earlier access turned out to be to the same address, and the later access is not a read of the same write, then the later access must be restarted to avoid coherence violations.

MP.RTf.inv+dmb+ctrl		AArch64
Initial state: x -> invalid, z -> pa1, *pa1 = 1, y -> pa2, 0:X0=desc3(z), 0:X1=pte3(x), 0:X2=1, 0:X3=y, 1:X1=y, 1:X3=x		
Thread 0	Thread 1	Thread1 EL1 Handler
STR X0, [X1] DMB SY STR X2, [X3]	LDR X0, [X1] CBNZ X0, L0 L0: LDR X2, [X3]	MOV X2, #0 MRS X13, ELR_EL1 ADD X13, X13, #4 MSR ELR_EL1, X13 ERET
Allowed: 1:X0 = 1 & 1:X2=0		

The non-TLB read in Thread 1 (e1) is not locally ordered after the earlier load (d), despite the control dependency. This is because the processor can speculatively perform the translation table walk, before the earlier read is satisfied.

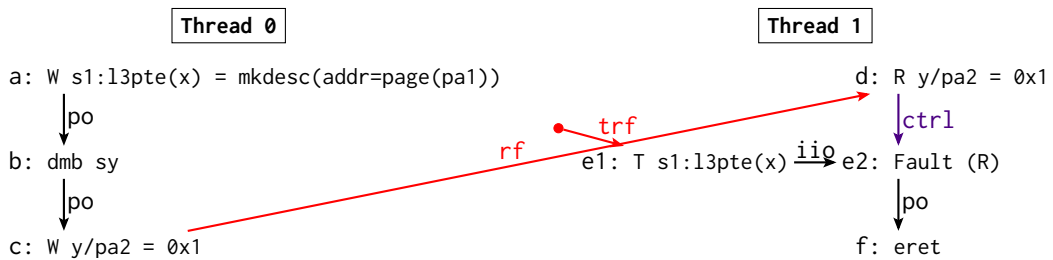


Figure 8.21: Test MP.RTf.inv+dmb+ctrl

MP.RT.inv+dmb+ctrl-trfi		AArch64
Initial state: w -> invalid, x -> pa1, *pa1 = 0, y -> pa2, 0:X0=1, 0:X1=x, 0:X2=1, 0:X3=y, 1:X1=y, 1:X2=mkdesc3(oa=pa1), 1:X3=pte3(w), 1:X5=w		
Thread 0	Thread 1	Thread1 EL1 Handler
STR X0, [X1] DMB SY STR X2, [X3]	LDR X0, [X1] CBZ X0, LC00 LC00: STR X2, [X3] LDR X4, [X5]	MOV X4, #2 MRS X13, ELR_EL1 ADD X13, X13, #4 MSR ELR_EL1, X13 ERET
Forbidden: 1:X0 = 1 & 1:X4=0		

The non-TLB read of the translation table entry (f1) cannot read from a forwarded thread-local write (event e) when on a speculative path, requiring that f1 be ordered after d.

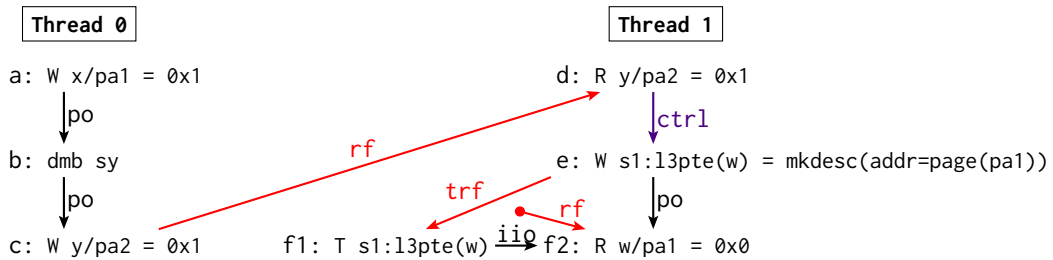


Figure 8.22: Test MP.RT.inv+dmb+ctrl-trfi

CoTtF.inv+po		AArch64
Initial state: x -> invalid, y -> pa1, *pa1 = 1, 0:X0=desc3(y), 0:X1=pte3(x), 1:X1=x, 1:X3=x		
Thread 0	Thread 1	Thread1 EL1 Handler
STR X0, [X1]	LDR X2, [X1] MOV X0, X2 LDR X2, [X3]	MOV X2, #0 MRS X13, ELR_EL1 ADD X13, X13, #4 MSR ELR_EL1, X13 ERET
Allowed: 1:X0 = 1 & 1:X2=0		

The translation-table-walks of the two same-address loads of x can execute out-of-order, even when the later translation table read (c1) reads from a different write than the program-order-earlier one (b1).

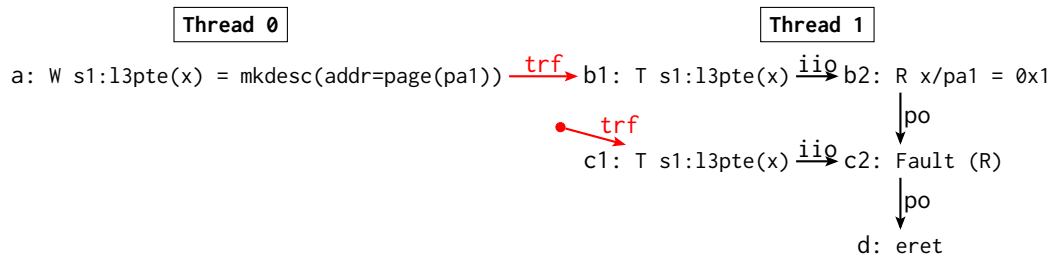


Figure 8.23: Test CoTtF.inv+po

3446 Translation table walk reads, while they are reads, are not required to be restarted to recover coherence,
 3447 do not do this hazard checking. This is most obvious in the [CoTtF.inv+po](#) (Figure 8.23, p.127), where
 3448 the two translations for the two same-address loads in Thread 1 are performed out-of-order.

3449 8.4.6 Single-copy atomicity

3450 In the base memory model, there are two key guarantees on the *atomicity* of reads and writes: single-copy
 3451 atomicity and multi-copy atomicity.

3452 Recall that single-copy atomic reads always read the maximum they can from another single-copy atomic
 3453 write; in particular a 64-bit atomic read never partially reads from another 64-bit atomic write.

3454 Translation table walk reads are 64-bit single-copy-atomic reads of memory. This means that each of the
 3455 reads generated by a translation table walk will read the entire descriptor in one shot. This causes the
 3456 [CoWroW.inv+dsb-isb](#) test (Figure 8.24, p.128) to be forbidden, disallowing reading the output address
 3457 obtained from one write, and access permissions from another.

3458 8.4.7 Multi-copy atomicity

3459 Multi-copy atomicity is a guarantee that requires any update to memory to propagate to all other threads
 3460 simultaneously. This is one of the core guarantees Arm and RISC-V give, but earlier versions of Arm and
 3461 IBM's Power architectures do not. On Arm, threads can observe their own writes early, through write
 3462 forwarding, giving a weaker form of multi-copy atomicity referred to as *other-multi-copy atomicity* by
 3463 Arm.

3464 Microarchitecturally, a thread can only read another thread's write by reading from a global coherent
 3465 storage subsystem. This ensures that after the thread reads from that write, any other thread must also
 3466 see that write, or something coherence after it. While this is a property that the base model seems to
 3467 have, whether it is true for accesses during translation table walks is a separate question.

3468 The non-TLB reads during a translation table walk respect other-multi-copy-atomicity: if one other
 3469 thread has observed a write through a translation table walk, then future translation table walk non-TLB
 3470 reads by other threads will also observe that write (or something newer). Axiomatically, if one thread
 3471 translation-reads-from a write, then all translation-table-walk reads locally-ordered after another memory
 3472 event, which is itself ordered after the other thread's translation-table-walk read, will be ordered after
 3473 that translation-table-walk read.

CoWroW.inv+dsb-isb		AArch64
Initial state: x -> invalid, *pa1 = 0, 0:X0=mkdesc3(oa=pa1, AP=3), 0:X1=pte3(x), 0:X2=1, 0:X3=x		
Thread 0	Thread0 El1 Handler	
STR X0, [X1] DSB SY ISB STR X2, [X3]	MRS X20,ELR_EL1 ADD X20,X20,#4 MSR ELR_EL1,X20 ERET	
Forbidden: *pa1=1		

The translation table walk of the second store must read from the entire write from the earlier store, or not at all, forbidding its translation walk from reading a mix of both the initial state and the earlier write. This means there should be no way the final store can happen, as it must either be invalid or read-only. Note that isla does not generate candidates with non-atomic reads which are supposed to be single-copy atomic, so there is no generated events diagram for this test.

Figure 8.24: Test CoWroW.inv+dsb-isb

3474 There are three combinations of multi-thread reads of interest, where a weaker architecture (with split
3475 pagetable and data memory storage) might have mixed non-multi-copy atomic behaviours. The first
3476 of these is the most basic: translation-read to translation-read, that is, the pagetable accesses are
3477 multi-copy atomic, and this is what forbids reading the old translation table value in Thread 2 in
3478 the [WRC.TRTf.inv+po+dsb-isb](#) test (Figure 8.25, p.129). The other two are combinations of read-to-
3479 translation-read and translation-read-to-read; these show us that translation accesses and explicit data
3480 accesses are architecturally unified: information about the memory state learned through one kind of access
3481 applies to accesses of the other. This is what forbids the [WRC.RRTf.inv+dmb+dsb-isb](#) (Figure 8.26, p.130)
3482 and [WRC.TRR.inv+po+dsb](#) (Figure 8.27, p.130) tests from reading the old value from memory at the
3483 end.

3484 **8.4.8 Translation-table-walk intra-walk ordering**

3485 All the tests so far have been concerned with changes to at most one of the translation table entries during
3486 a single walk. However, as we saw in Chapter 7, each translation table walk performs many reads, as
3487 many as 24, for a single translation.

3488 The ASL for the translation table walker performs each translation, in order, starting with the root,
3489 and ending with the leaf entry. While reads in a thread can be executed out-of-order, translation-reads
3490 within a translation table walk cannot, as this would require the hardware to do value speculation on the
3491 next-level table address, but as discussed in §8.4.5, using speculative values in a walk is forbidden.

3492 Requiring the translation reads from a translation table walk to be satisfied in translation walk order has
3493 an observable effect. For example, in the [ROT.inv+dsb](#) test (Figure 8.28, p.131), the translation table
3494 walk of the read in Thread 1 must see the writes to the translation table done by Thread 0 in the order
3495 they were propagated out to memory, and so reading from the old level 3 entry is forbidden.

3496 **8.4.9 Multiple translations within a single instruction**

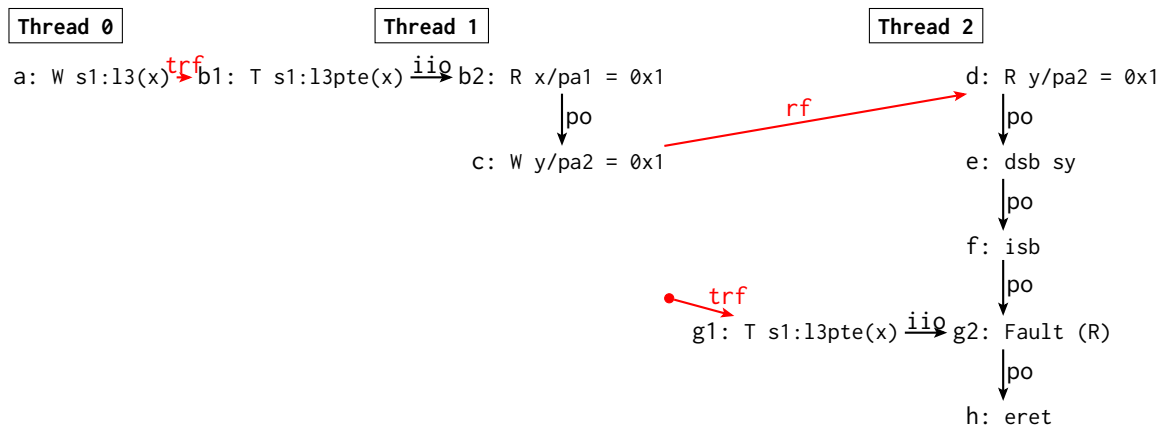
3497 Some instructions generate multiple explicit memory events, for example, the ‘load pair’ and ‘store pair’
3498 instructions, misaligned accesses, and read-modify-writes. When there are multiple explicit memory events,
3499 there will be a dedicated translation for each of them, with its own translation table walk.

3500 Here, the architecture as it is written today is overly sequentialised: the ASL for these cases performs
3501 each translation (and the respective access) in some order, but the architectural intent is that the separate
3502 translations should be unordered with respect to each other.

3503 Misaligned accesses, and the load pair and store pair instructions, should generate explicit memory events
3504 and associated translations which are unordered with respect to each other.

WRC.TRTf.inv+po+dsb-isb AArch64

Initial state: x -> invalid, z -> pa1, *pa1 = 1, y -> pa2, 0:X0=desc3(z), 0:X1=pte3(x), 1:X1=x, 1:X2=1, 1:X3=y, 2:X1=y, 2:X3=x				
Thread 0	Thread 1	Thread 2	Thread1 EL1 Handler	Thread2 EL1 Handler
STR X0, [X1]	LDR X0, [X1] STR X2, [X3]	LDR X0, [X1] DSB SY ISB LDR X2, [X3]	MOV X0, #0 MRS X13, ELR_EL1 ADD X13, X13, #4 MSR ELR_EL1, X13 ERET	MOV X2, #0 MRS X13, ELR_EL1 ADD X13, X13, #4 MSR ELR_EL1, X13 ERET
Forbidden: 1:X0=1 & 2:X0=1 & 2:X2=0				



The translation-read of x (g1) is ordered after another translation-read of the same address x (b1), so by multi-copy-atomicity g1 may not read from an older write than b1 did.

Figure 8.25: Test WRC.TRTf.inv+po+dsb-isb

WRC.RRTf.inv+dmb+dsb-isb AArch64

Initial state: x |-> invalid, z |-> pa1, *pa1 = 1,
y |-> pa2, 0:X0=desc3(z), 0:X1=pte3(x),
1:X1=pte3(x), 1:X2=1, 1:X3=y,
2:X1=y, 2:X3=x

Thread 0	Thread 1	Thread 2	Thread2 EL1 Handler
STR X0, [X1]	LDR X0, [X1] DSB SY STR X2, [X3]	LDR X0, [X1] DSB SY ISB LDR X2, [X3]	MOV X2, #0 MRS X13, ELR_EL1 ADD X13, X13, #4 MSR ELR_EL1, X13 ERET

Forbidden: 1:X0=desc3(z) & 2:X0=1 & 2:X2=0

The translation-read of x (h1) is ordered after the read of the translation table entry (b) and so by multi-copy-atomicity it cannot read from an older write than b did. The dsb-isb sequence in Thread 2 ensures the translation-table-walk of g is ordered after the program-order earlier read even without FEAT_ETS (see §8.4.3).

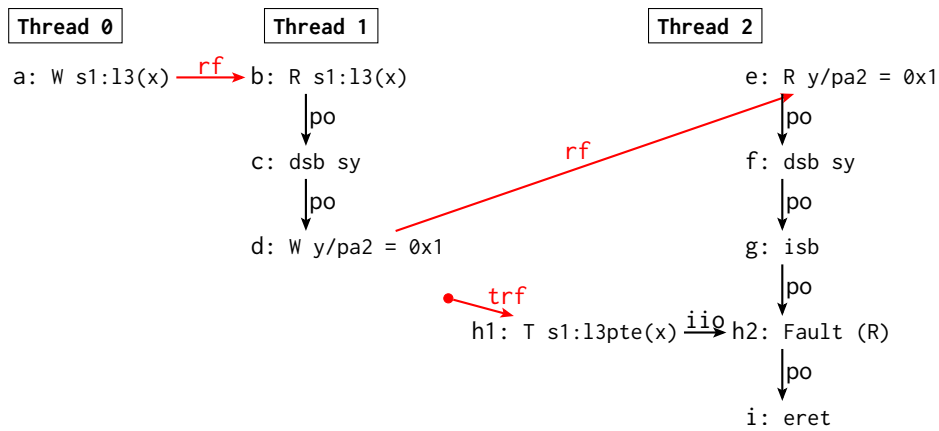


Figure 8.26: Test WRC.RRTf.inv+dmb+dsb-isb

WRC.TRR.inv+po+dsb AArch64

Initial state: x |-> invalid, y |-> pa2, *pa1 = 1,
0:X0=mkdesc3(oa=pa1), 0:X1=pte3(x), 1:X0=0, 1:X1=x,
1:X2=1, 1:X3=y, 2:X1=y, 2:X3=pte3(x)

Thread 0	Thread 1	Thread 2	Thread1 EL1 Handler
STR X0, [X1]	LDR X0, [X1] STR X2, [X3]	LDR X0, [X1] DSB SY LDR X2, [X3]	MRS X13, ELR_EL1 ADD X13, X13, #4 MSR ELR_EL1, X13 ERET

Forbidden: 1:X0=1 & 2:X0=1 & 2:X2=0

The read of the translation table entry for x (f) is ordered after the translation read of x (b1) and so by multi-copy-atomicity it cannot read from an older write than b1 did. The dsb in Thread 2 is sufficient to order the reads, any preserved read-to-read thread-local ordering suffices.

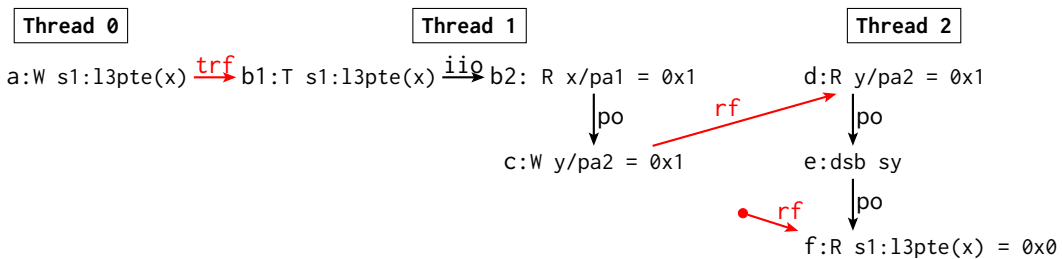
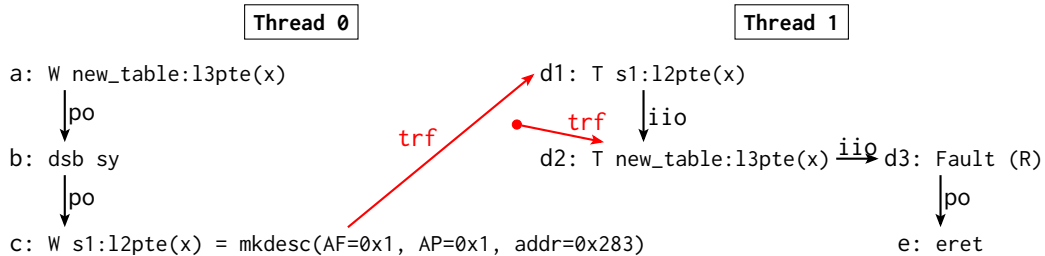


Figure 8.27: Test WRC.TRR.inv+po+dsb

Initial state: ipa1 -> pa1, x -> invalid at level 2, s1table new_table 0x280000 {, x -> invalid, }, 0:X0=mkdesc3(oa=ipa1), 0:X1=pte3(x, new_table), 0:X2=mkdesc2(table=0x283000), 0:X3=pte2(x), 0:PSTATE.EL=1, 1:X1=x		
Thread 0	Thread 1	Thread1 EL1 Handler
STR X0,[X1] DSB SY STR X2,[X3]	LDR X0,[X1]	<pre>// read ESR_EL1.ISS, to see if fault at Level 2 or 3. MRS X14,ESR_EL1 AND X14,X14,#7 CMP X14,#7 MOV X17,#1 MOV X18,#2 // if ESR_EL1.ISS.DFSC == Translation Level 3 // then x0 = 1 else x0 = 2 CSEL X0,X17,X18,eq // advance ELR MRS X13,ELR_EL1 ADD X13,X13,#4 MSR ELR_EL1,X13 // return ERET</pre>
Forbidden: 1:X0=1		



The translation-table walk from the read of x in Thread 1 must perform its translation non-TLB reads, $d1$ and $d2$, in the order they appear in the walk, forbidding reading from the new level 2 table entry in $d1$, but then reading the stale initial value for that entry from memory. The test listing contains some concrete values to make it executable in isla: the location of the new table is fixed at page 280, so that it is not symbolic; the location of the level 3 table within the new tree will be in page 283, which is known from the fixed isla configuration of page tables. Whether the exception comes from the level 2 or the level 3 entry can be determined by reading the ISS field of the ESR_EL1 register, which is what the exception handler does in this test.

Figure 8.28: Test ROT.inv+dsb

CoWinvT+po		AArch64
Initial state: $x \mapsto pa1, 0:X0=0, 0:X1=pte3(x), 0:X3=x$		
Thread 0	Thread0 El1 Handler	
STR X0, [X1] LDR X2, [X3]	MOV X2, #1 MRS X13, ELR_EL1 ADD X13, X13, #4 MSR ELR_EL1, X13 ERET	
Allowed: $0:X2 = 0$		

The translation read (b1) of the last-level entry for x can be executed out-of-order with respect to the program-order earlier store (a) of an invalid entry to $pte3(x)$.

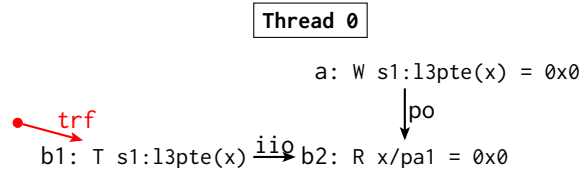


Figure 8.29: Test CoWinvT+po

8.5 Caching of translations in TLBs

We have seen in §8.4 that, while non-TLB reads do not necessarily preserve the program-order without additional synchronisation, due to the out-of-order execution of instructions, those translation table reads get satisfied from the coherent storage subsystem or from forwarding from earlier stores, much like the normal explicit data reads do. This section will explore what happens when translation table walk reads may instead be satisfied from the TLB.

Unfortunately for the programmer, the TLB need not be coherent with memory: it can have stale values. This section explores the behaviours that arise from this caching of stale values.

8.5.1 Cached translations

In the previous section we carefully constructed tests which began with an initially invalid translation, to avoid TLB caching issues. Here, we will generally start with entries that are valid, and so might be present in the TLB.

The following CoWinvT+po test (Figure 8.29) begins with an initially valid (and therefore potentially initially cached in the TLB) translation for the virtual address x . It then updates the last-level translation table entry for x , setting it to 0 , making it invalid (and thus unmapping x). Then, program order later, the same thread tries to read x .

The read can succeed, as its translation can read from the old value from memory. We saw earlier that translation table walks can be executed out-of-order with respect to program order (§8.4.1), but even inserting thread-local ordering to the translation, such as in test CoWinvT+dsb-isb (Figure 8.30, p.133), does not forbid it.

CoWinvT+dsb-isb		AArch64
Initial state: $x \mapsto pa1, 0:X0=0, 0:X1=pte3(x), 0:X3=x$		
Thread 0	Thread0 El1 Handler	
STR X0, [X1] DSB SY ISB LDR X2, [X3]	MOV X2, #1 MRS X20, ELR_EL1 ADD X20, X20, #4 MSR ELR_EL1, X20 ERET	
Allowed: $0:X2 = 0$		

The translation read (d1) of the last-level entry for x is required to be satisfied after the earlier store of the invalid entry (a) because of the intervening `dsb sy; isb` sequence, but the value could come from a cached value in the TLB, allowing d1 to read from a stale value.

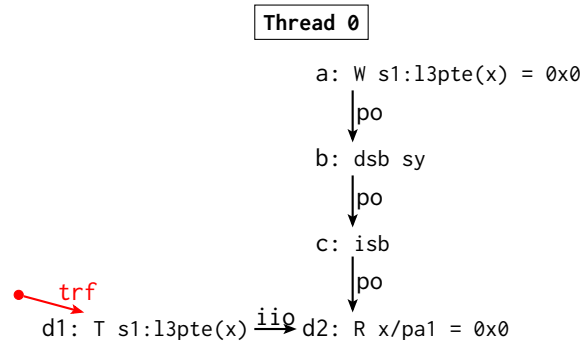


Figure 8.30: Test CoWinvT+dsb-isb

8.5.2 TLB fills

3525

3526 Translation table walks can be requested by the core in two different ways: (1) through the architectural
3527 execution of an instruction; or, (2) from a spontaneous translation table walk (for example, due to
3528 speculation and prefetching of data or instructions). In either case, the result of that walk can be cached
3529 in the TLB and recalled for other translation table walks.

3530 Architecturally, a TLB fill is no different to a normal translation table walk. Each TLB fill originates
3531 from a non-TLB read, with all the behaviours described in the previous sections. Later translation table
3532 walks are allowed, however, to recall an earlier value and then reuse that rather than doing a fresh read.

3533 **Spontaneous walks** The hardware may, at any time, prefetch or speculatively read some address. These
3534 appear as spontaneous translation table walks. Those spontaneous walks may be cached. We can see this
3535 occurring in the following `MP.RT.inv+poloc-dmb+ctrl-isb` test (Figure 8.31, p.134), where a spontaneous
3536 translation and the resulting TLB fill allows a future translation table walk to see a stale value.

3537 **Speculative paths** Since translation table walks, and therefore TLB fills from the result of those walks,
3538 can happen at any point, there is no need to consider TLB fills of architectural translation table walks
3539 down speculative paths as any such behaviour is subsumed by a spontaneous fill.

3540 However, as described earlier, we saw that writes cannot be forwarded to translation table walks when
3541 down speculative paths (§8.4.5). This naturally excludes TLB fills of still speculative writes: since a
3542 speculative write cannot be used in the result of a translation table walk, it cannot end up cached in a
3543 TLB.

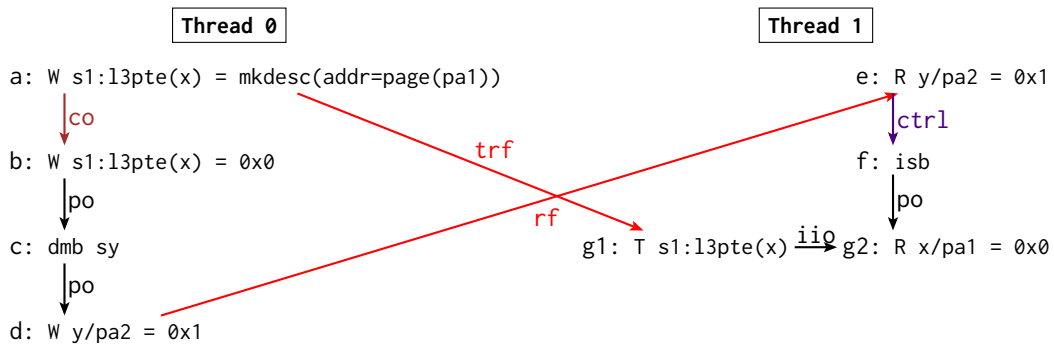
8.5.3 microTLBs

3544

3545 So far we have spoken as if entries are, at any particular moment in time, either present in the TLB or not.
3546 Hardware, however, may have multiple *micro*TLBs for the same thread, each with their own potential
3547 cached entry for the same address.

3548 In effect, these microTLBs behave as if they were a larger non-deterministic TLB with potentially many
3549 values for each entry. The presence of these smaller caching structures in a superscalar machine means

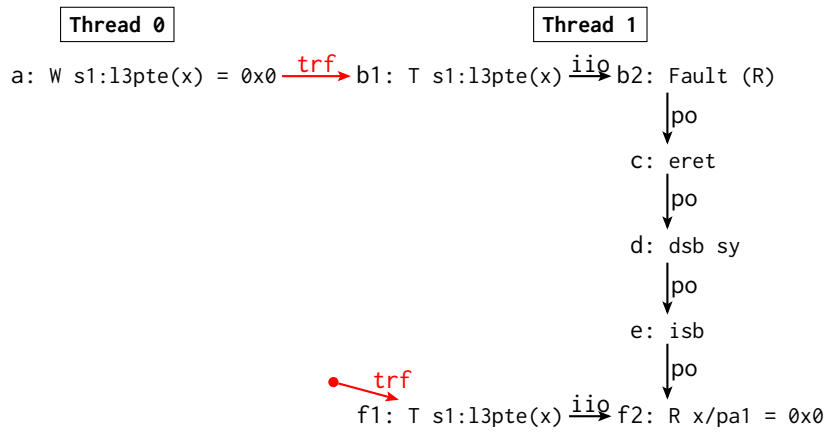
MP.RT.inv+poloc-dmb+ctrl-isb		AArch64
Initial state: x -> invalid, y -> pa2, *pa1 = 0, *pa2 = 0, 0:X0=mkdesc3(oa=pa1), 0:X1=pte3(x), 0:X2=0, 0:X3=pte3(x), 0:X4=1, 0:X5=y, 1:X1=y, 1:X3=x		
Thread 0	Thread 1	Thread1 EL1 Handler
STR X0, [X1] STR X2, [X3] DMB SY STR X4, [X5]	LDR X0, [X1] CBNZ X0, L0 L0: ISB MOV X2, #1 LDR X2, [X3]	MRS X13, ELR_EL1 ADD X13, X13, #4 MSR ELR_EL1, X13 ERET
Allowed: 1:X0 = 1 & 1:X2=0		



A spontaneous walk and fill can happen on Thread 1 after the write of the valid entry to pte3(x) (a), but before the immediate re-invalidation of that entry (b), allowing the later translation table walk (g1) to see the old cached entry, even though the architectural translation table walk could not have happened while the valid entry was visible.

Figure 8.31: Test MP.RT.inv+poloc-dmb+ctrl-isb

CoTfT+dsb-isb		AArch64
Initial state: x -> pa1, y -> pa1, *pa1 = 0, 0:X0=0, 0:X1=pte3(x), 1:X1=x, 1:X3=x		
Thread 0	Thread 1	Thread1 EL1 Handler
STR X0, [X1]	LDR X2, [X1] MOV X0, X2 DSB SY ISB LDR X2, [X3]	MOV X2, #1 MRS X13, ELR_EL1 ADD X13, X13, #4 MSR ELR_EL1, X13 ERET
Allowed: 1:X0 = 1 & 1:X2=0		



The earlier translation read (b1) reads from the new invalid entry, reading from memory (as it cannot have been in the TLB), but a later translation read (f1) of the same location can still potentially see a stale cached entry.

Figure 8.32: Test CoTfT+dsb-isb

3550 that different instructions may be accessing different TLBs at the same time. This allows later instructions
3551 to ‘skip’ over a previously seen cached entry, and then see it again later.

3552 These effects can be seen in the [CoTfT+dsb-isb](#) test (Figure 8.32), where the presence of these micro-TLBs
3553 (or other distributed caching structures) permits later events (even locally-ordered later) to see old cached
3554 entries after earlier events witnessed a TLB miss.

3555 **Break-before-make and restrictions** We will see later that the ability to have multiple cached entries
3556 for a single address causes problems for software managing coherence, and imposes extra restrictions on
3557 software practice. This means that, in general, the effects of the micro-TLBs are restricted to only those
3558 combinations that do not cause *break-before-make violations* (see §8.6.5).

3559 8.5.4 Partial caching of walks

3560 TLBs need not cache entire virtual to physical translations. Instead, they are free to cache any subset of
3561 the reads from the walk separately.

3562 **Caching up to last-level table** The most common kind of caching structure we are aware of in mi-
3563 croarchitecture is the walk cache (see §8.3.1). Traditionally, a TLB would store entire virtual to physical
3564 mappings, making it fast to lookup the translation (often a single cycle), but there was limited space,
3565 and this induced heavy burden on a TLB miss or TLB invalidation. Walk caches store the last-level
3566 table entry, allowing TLB invalidation of leaf entries and TLB misses to re-use a prefix of the walk and

3567 perform a minimal number of accesses. This can be seen in the [MP.RTT.inv3+dmb-dmb+dsb-isb](#) test
3568 (Figure 8.33, p.137), where a walk cache allows the table entry to be cached separately from the last-level
3569 entry, allowing the last translation read to read from a much newer value.

3570 **Independent caching of IPAs** In a two-stage regime, the virtual addresses are first translated into
3571 intermediate physical address. The secondary translations based on the intermediate physical addresses,
3572 either of the final output address or of any of the intermediate table addresses, may be cached in the TLB
3573 without remembering the originating virtual address. This means that these cached translations may be
3574 recalled for translations of different virtual addresses.

3575 In addition, pre-fetching may perform translations of arbitrary IPAs. This means that any cached
3576 translations might not correspond to any valid whole translation table walk, but may still be used during
3577 such walks. This is most clear in [ROT.invs1+dmb2](#) (Figure 8.34, p.138), where, although the IPA was
3578 never reachable from the stage 1 translations, the old IPA to PA mapping was cached and used later.

3579 **Caching of whole translation** A common configuration for the TLB is to cache whole translation walks,
3580 from virtual to physical. This kind of caching has an important caveat: there is no requirement for the
3581 TLB to remember the intermediate physical address of any stage 2 translations that were done during the
3582 walk, including the final stage 2 walk of the access address itself. This means that TLB invalidations by
3583 IPA might not remove all the cached data associated with a cached entry for that IPA, if there is a whole
3584 cached translation which is derived from that entry. See §8.6.4 for more discussion on how this affects
3585 requirements on software.

3586 **Caching of individual entries** Architecturally, Arm wish to allow many more implementations of TLBs
3587 and translation caching structures than currently known hardware contains.

3588 The weakest variation on this is allowing each individual translation table entry to be cached separately
3589 and independently. One could construct litmus tests for each of the possible combination of translation
3590 table entries, but there would be overwhelmingly many of these, or even a ‘most relaxed’ version where
3591 every translation table entry comes from different previous translations, but this would be too large to
3592 show here. So, for simplicity we show just one of them here, [ROT.inv2+dmb](#) (Figure 8.35, p.139), where
3593 the last-level entry came from a newer value than the previous levels.

3594 **8.6 TLB maintenance**

3595 Recovering coherence for translation reads in the presence of TLB caching can be achieved through the
3596 use of TLB *maintenance* instructions: namely, the TLBI (‘TLB invalidate’) family of instructions.

3597 TLB maintenance generally performs two microarchitectural effects: erasing stale entries from the TLB,
3598 ensuring future TLB fills (for example, due to a translation read) will see the coherent value from memory;
3599 and the more subtle effect of discarding any partially executed instructions, on other cores, which had
3600 already begun execution using a stale entry but had not yet finished executing. We now explore both of
3601 these effects, and the subtle interaction with other parts of the VMSA, in more detail.

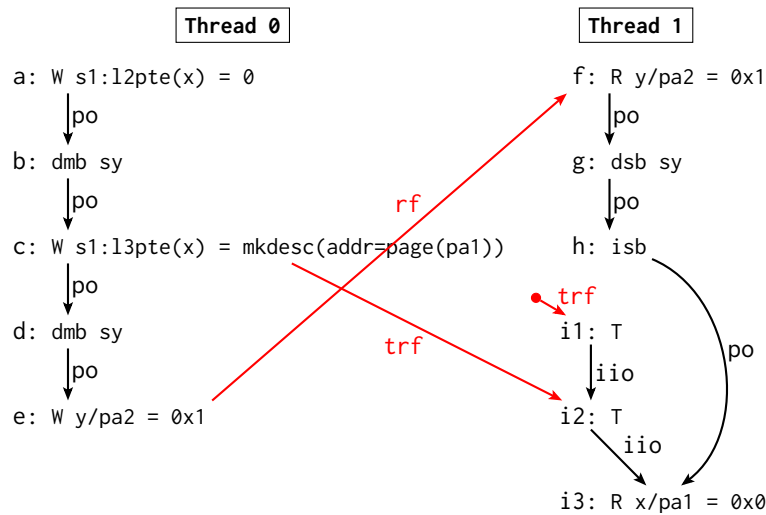
3602 **8.6.1 Recovering coherence**

3603 We saw in §8.5.1 that stale values cached in the TLB can cause coherence violations in the translation,
3604 for example, in the [CoWinvT+dsb-isb](#) test (Figure 8.30, p.133). By inserting the correct TLBI sequence
3605 into that test, we produce a new test, [CoWinvT.EL1+dsb-tlbi-dsb-isb](#) (Figure 8.36, p.140), which is now
3606 forbidden.

3607 There are many flavours of TLBI that could have been inserted into this test. The one in the figure is
3608 TLBI VAE1: TLB invalidation by virtual address, for the EL1&0 translation regime. Using a TLBI-by-VA
3609 means the programmer has to provide the virtual page to invalidate, and the TLBI only affects addresses
3610 for that specific invalidated entry, not all of them.

3611 Using the incorrect TLBI leads to insufficient invalidation occurring. For example, in the aforementioned
3612 [CoWinvT.EL1+dsb-tlbi-dsb-isb](#) test (Figure 8.36, p.140), if the TLBI had the wrong page then it would
3613 have no effect and the test would remain allowed.

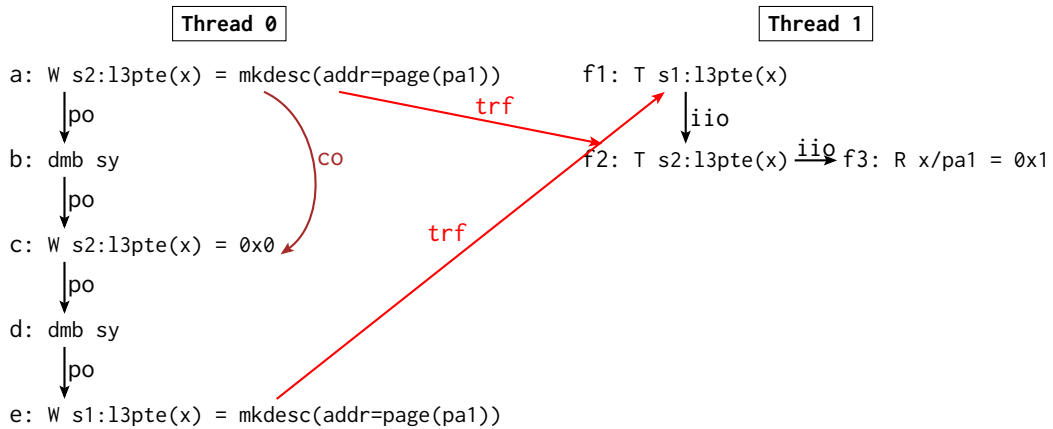
Initial state: assert $x[48..21] \neq y[48..21]$, $x \mapsto \text{invalid}$, $y \mapsto \text{pa2}$, $*\text{pa1} = 0$, $*\text{pa2} = 0$, $0:\text{X0}=0$, $0:\text{X1}=\text{pte2}(x)$, $0:\text{X2}=\text{mkdesc3}(\text{oa}=\text{pa1})$, $0:\text{X3}=\text{pte3}(x)$, $0:\text{X4}=1$, $0:\text{X5}=y$, $1:\text{X1}=y$, $1:\text{X3}=x$		
Thread 0	Thread 1	Thread1 EL1 Handler
STR X0, [X1] DMB SY STR X2, [X3] DMB SY STR X4, [X5]	LDR X0, [X1] DSB SY ISB MOV X2, #1 LDR X2, [X3]	MRS X13, ELR_EL1 ADD X13, X13, #4 MSR ELR_EL1, X13 ERET
Allowed: $1:\text{X0} = 1$ & $1:\text{X2}=0$		



The translation-read of the level 2 entry for x ($i1$) can read from stale writes from a translation that the subsequent level 3 translation-read ($i2$) does not read from, as the level 2 entry could have been cached in the TLB (in this case, a co-located ‘walk cache’ structure), while the level 3 entry gets read from memory. In the test, x is initially invalid at level 3, and x and y have different level 2 entries (by ensuring they are not in the same 2 MiB region), and writes zero to the level 2 entry for x (a) and then overwrites the previously-zero level 3 entry to point to pa1 , such that the final read of x could only see a valid entry if the walk read-from the new level 3 entry, but a stale cached level 2 entry. The magic numbers are concrete instantiations from *isla-axiomatic*’s symbolic evaluation.

Figure 8.33: Test MP.RTT.inv3+dmb-dmb+dsb-isb

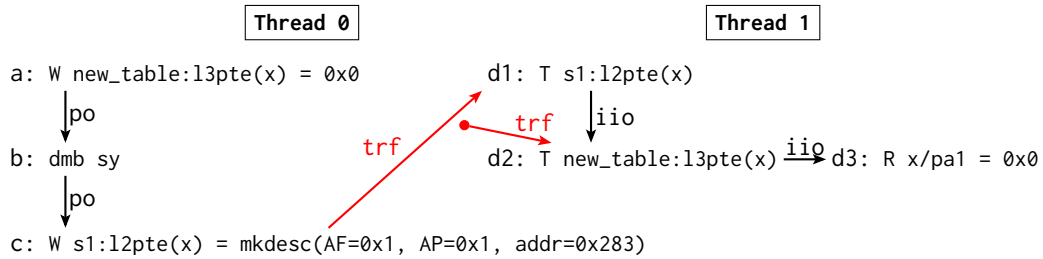
Initial state: intermediate ipa1, x -> invalid at level 2, ipa1 -> pa1, *pa1 = 1, 0:X0=mkdesc3(oa=pa1), 0:X1=pte3(x, s2_page_table_base), 0:X2=0, 0:X3=pte3(x, s2_page_table_base), 0:X4=mkdesc3(oa=ipa1), 0:X5=pte3(x), 0:PSTATE.EL=1, 1:X1=x			
Thread 0	Thread 1	Thread1 EL1 Handler	Thread1 EL2 Handler
STR X0, [X1] DMB SY STR X2, [X3] DMB SY STR X4, [X5]	MOV X0, #0 LDR X0, [X1]	MRS X13, ELR_EL1 ADD X13, X13, #4 MSR ELR_EL1, X13 ERET	MRS X13, ELR_EL2 ADD X13, X13, #4 MSR ELR_EL2, X13 ERET
Allowed: 1:X0=1			



The translation read of the stage 2 leaf entry for x (f2) can read from an old cached version, from the write (a) even though it was not reachable by any translation table walk for any VA (as the IPA it maps was not mapped by any stage 1 tables before it was overwritten by (b)). This test relies on translation table walks being naturally ordered (by iio), see §8.4.8.

Figure 8.34: Test ROT.invs1+dmb2

Thread 0		Thread 1	Thread1 EL1 Handler
Initial state: intermediate ipa1, assert pa1 == ipa1, ipa1 -> pa1, x -> invalid at level 2, sitable new_table 0x280000 { x -> ipa1 }, 0:X0=0, 0:X1=pte3(x, new_table), 0:X2=mkdesc2(table=0x283000), 0:X3=pte2(x), 0:PSTATE.EL=1, 1:X1=x			
STR X0, [X1] DMB SY STR X2, [X3]		MOV X0, #1 LDR X0, [X1]	MRS X13, ELR_EL1 ADD X13, X13, #4 MSR ELR_EL1, X13 ERET
Allowed: 1:X0=0			



The translation-read of the level 3 entry (d2) can read from a stale cached translation, which was cached before the write to the level 2 entry (c). Note that this test assumes that the original `new_table` was reachable (and therefore could be cached) before the write c. See §8.8.1 for a discussion on this.

Figure 8.35: Test ROT.inv2+dmb

3614 FEAT_nTLBPA

3615 Armv8.4-A introduced a new optional Arm feature, FEAT_nTLBPA [12, A2.2.1 (p79)].

3616 This feature adds a field to the memory model feature register (AA64MMFR1_EL1) which identifies whether
3617 the current processor's TLB may contain non-coherent copies of stage 1 entries indexed by those entries'
3618 intermediate physical address. Microarchitecturally, this corresponds to there being non-coherent caches
3619 associated with the TLB, which must be flushed on a TLBI. These caches would allow TLB misses to
3620 read from a non-coherent cache, thus not seeing the most up-to-date value from the coherent storage
3621 subsystem like described in §8.4.

3622 This change adds a field to the register, whose reserved value in Armv8.0 corresponds to non-coherent
3623 caches existing. This implies that in processors without FEAT_nTLBPA, one should assume that TLBs may
3624 contain non-coherent caching structures, including prior to the introduction of the FEAT_nTLBPA feature
3625 entirely: it is not clear to us whether this is intentional. Therefore, some behaviours described here may
3626 assume a setting that is too strong, erroneously assuming all non-TLB translation-reads read from the
3627 coherent-latest write. Our experimental data did not include any devices with incoherent non-TLB reads.

3628 8.6.2 Thread-local ordering and TLBI

3629 TLB maintenance instructions are not naturally locally ordered with respect to other instructions in the
3630 instruction stream. This means that they can be executed out-of-order with respect to other instructions.
3631 To ensure they are synchronized with other instructions, the programmer can use the DSB barrier instruction
3632 to impose order on the instructions before and after it.

3633 Leaving out one or both of the DSBs around the TLBI leads to insufficient ordering around the TLBI, and
3634 allows the invalidation to occur at the wrong time. For example, the `CoWinvT.EL1+tlbi-dsb-isb` test
3635 (Figure 8.37, p.141), a version of Figure 8.36 without the DSB b, is allowed as the initial write and TLBI
3636 may be re-ordered, negating the architectural effect of the TLBI.

CoWinvT.EL1+dsb-tlbi-dsb-isb		AArch64
Initial state: $x \mapsto pa1$, $0:X0=0$, $0:X1=pte3(x)$, $0:X3=x$, $0:X5=page(x)$, $0:PSTATE.EL=1$		
Thread 0	Thread0 EL1 Handler	
STR X0, [X1] DSB SY TLBI VAE1, X5 DSB SY ISB LDR X2, [X3]	MOV X2, #1 MRS X13, ELR_EL1 ADD X13, X13, #4 MSR ELR_EL1, X13 ERET	
Forbidden: $0:X2 = 0$		

The translation-read of the translation table entry for x (f1) is required to happen after the earlier store (a), because of the intervening `dsb sy`; `isb` sequence (d and e), and cannot be satisfied from the TLB, because of the TLBI (c), forbidding it from still seeing a stale value. Note that TLBI instructions can only be executed from EL1 or above, so this test starts execution at EL1 rather than the usual default of EL0.

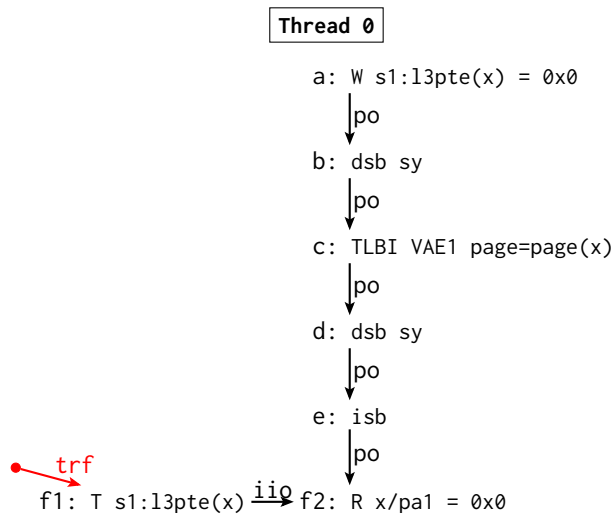


Figure 8.36: Test CoWinvT.EL1+dsb-tlbi-dsb-isb

CoWinvT.EL1+tlbi-dsb-isb		AArch64
Initial state: x -> pa1, 0:X0=0, 0:X1=pte3(x), 0:X3=x, 0:X5=page(x), 0:PSTATE.EL=1		
Thread 0	Thread0 EL1 Handler	
STR X0, [X1] TLBI VAE1, X5 DSB SY ISB LDR X2, [X3]	MOV X2, #1 MRS X13, ELR_EL1 ADD X13, X13, #4 MSR ELR_EL1, X13 ERET	
Final state: 0:X2 = 0		

The TLBI (b) can be re-ordered with program-order earlier events, due to the lack of DSBs ordering it after them, allowing the store (a) to happen later, letting the final translation read (e1) still see the old stale translation.

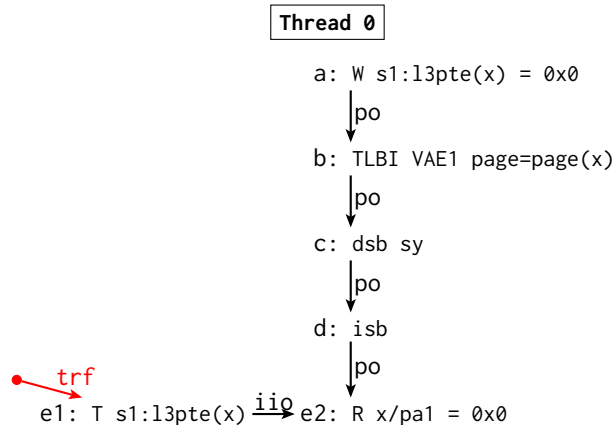


Figure 8.37: Test CoWinvT.EL1+tlbi-dsb-isb

8.6.3 Broadcast

3637

3638 Arm provide broadcast variants of the TLBI instructions. These are generally suffixed with the letters IS
 3639 ('Inner-shareable') in the mnemonic. Broadcast TLBIs, sometimes referred to as TLB *shootdowns*, allow
 3640 one processor to perform maintenance on other cores' TLBs. This is in contrast to other systems, such as
 3641 for x86, and IBM's Power architecture, where maintenance of other cores must be achieved in software
 3642 through the use of purely thread-local invalidation instructions.

3643 **TLB invalidation on another core** One of the simplest examples of multi-core invalidation is a message
 3644 passing invalidation pattern, where the old entry is removed, and a message is sent to another core. This
 3645 can be seen in the [MP.RT.EL1+dsb-tlbiis-dsb+dsb-isb](#) test (Figure 8.38, p.142).

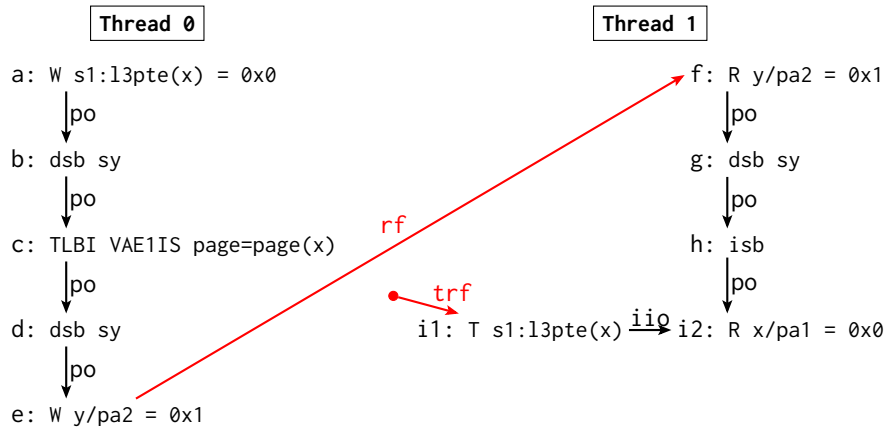
3646 **Instruction restarts** Broadcast TLBIs must do more than touch the other thread's TLB. If another
 3647 processor has an in-flight instruction which has started but not yet finished execution using a stale
 3648 translation, then that instruction must be restarted.

3649 This ensures that Arm broadcast TLBIs have the same behaviour as the traditional software IPI-based
 3650 shutdown (with context synchronization). It also provides a needed security guarantee: if a mapping is
 3651 taken away from a process, then future writes to the physical location it used to map to, should not be
 3652 visible to that process any more.

3653 This guarantee is captured in the [RBS+dsb-tlbiis-dsb](#) (Figure 8.39, p.143) (**Read-Broken-Secret**) test.
 3654 Once a mapping has been *broken*, and sufficient TLB maintenance performed, any future reads or writes
 3655 to the original physical location will not be visible through that mapping any more. Note, however, that
 3656 this does not mean that instructions which have already completed their execution will be restarted, even
 3657 if they occur after an earlier restarted instruction. This can be seen in the [RBS+dsb-tlbiis-dsb+poloc](#) test
 3658 (Figure 8.40, p.144), where the program-order later load can see the old value, even after the first faults.

3659 While here we describe things in terms of instruction restarting, these behaviours can be (and presumably
 3660 sometimes are) implemented in terms of waiting: instead of the TLBI forcibly restarting instructions that
 3661 already started but haven't finished, the TLBI can simply wait for them to complete.

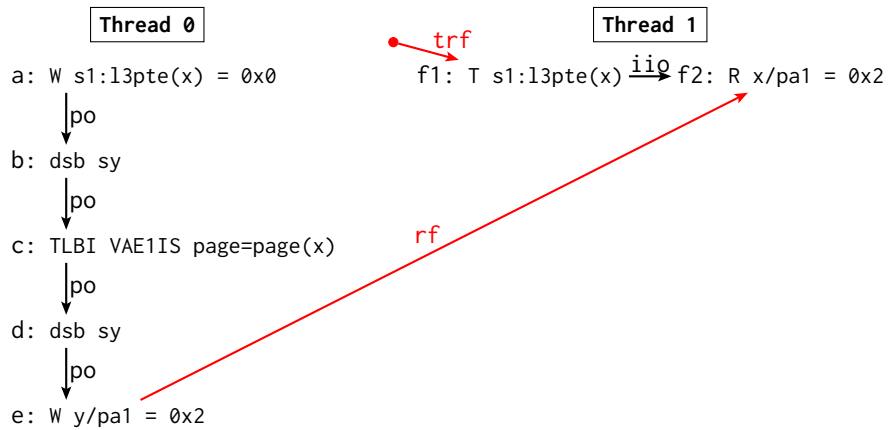
MP.RT.EL1+dsb-tlbiis-dsb+dsb-isb		AArch64
Initial state: x -> pa1, y -> pa2, 0:X0=0, 0:X1=pte3(x), 0:X2=1, 0:X3=y, 0:X4=page(x), 0:PSTATE.EL=1, 1:X1=y, 1:X3=x		
Thread 0	Thread 1	Thread1 EL1 Handler
STR X0, [X1] DSB SY TLBI VAE1IS,X4 DSB SY STR X2, [X3]	LDR X0, [X1] DSB SY ISB LDR X2, [X3]	MOV X2,#1 MRS X13,ELR_EL1 ADD X13,X13,#4 MSR ELR_EL1,X13 ERET
Forbidden: 1:X0 = 1 & 1:X2=0		



The broadcast TLBI on Thread 0 (c) ensures that the earlier unmapping (a) is seen by the ordered later translation read on Thread 1 (i1), by ensuring Thread 1's local TLB is cleaned of any stale entries for x.

Figure 8.38: Test MP.RT.EL1+dsb-tlbiis-dsb+dsb-isb

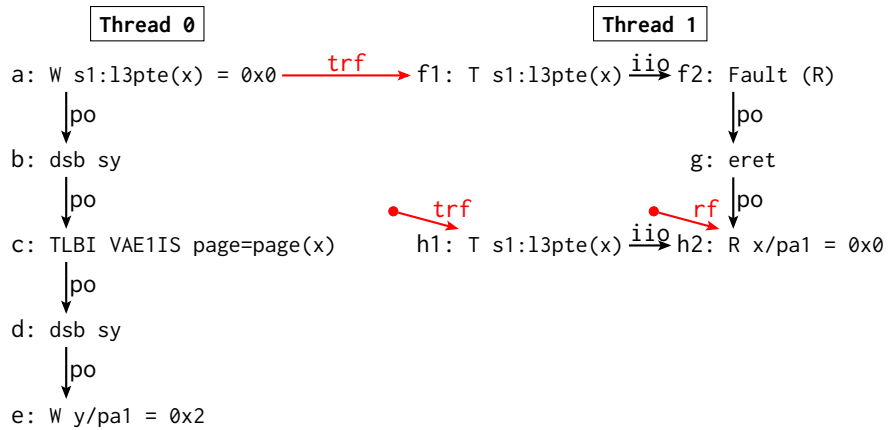
RBS+dsb-tlbiis-dsb		AArch64
Initial state: $x \mapsto pa1, y \mapsto pa1, *pa1 = 0, 0:X0=0, 0:X1=pte3(x), 0:X5=page(x), 0:X2=2, 0:X3=y, 0:PSTATE.EL=1, 1:X1=x$		
Thread 0	Thread 1	Thread1 EL1 Handler
STR X0, [X1] DSB SY TLBI VAE1IS, X5 DSB SY STR X2, [X3]	LDR X0, [X1]	MOV X0, #1 MRS X13, ELR_EL1 ADD X13, X13, #4 MSR ELR_EL1, X13 ERET
Forbidden: $1:X0 = 2$		



The broadcast TLBI of x (c) ensures that the execution of the load of x in Thread 1 either entirely executes using the old translation and finishes before the TLBI does, or begins execution after the TLBI finishes.

Figure 8.39: Test RBS+dsb-tlbiis-dsb

RBS+dsb-tlbiis-dsb+poloc		AArch64
Initial state: x -> pa1, y -> pa1, *pa1 = 0, 0:X0=0, 0:X1=pte3(x), 0:X5=page(x), 0:X2=2, 0:X3=y, 0:PSTATE.EL=1, 1:X1=x, 1:X3=x		
Thread 0	Thread 1	Thread1 EL1 Handler
STR X0, [X1] DSB SY TLBI VAE1IS, X5 DSB SY STR X2, [X3]	MOV X0, #1 LDR X0, [X1] MOV X2, #1 LDR X2, [X3]	MRS X13, ELR_EL1 ADD X13, X13, #4 MSR ELR_EL1, X13 ERET
Final state: 1:X0 = 1 & 1:X2 = 0		



Even though the broadcast TLBI on Thread 0 (c) ensures that not-yet-completed instructions using the old mapping are restarted, it does not require that the second load of x in Thread 1 (h) be restarted if it has already satisfied its value, as that value must have come from a write before the TLBI.

Figure 8.40: Test RBS+dsb-tlbiis-dsb+poloc

3662 **Atomic TLBIs** In previous tests, we describe behaviour in terms of writes that occur ‘before’ or ‘after’ a
3663 TLBI. Microarchitecturally, a TLBI instruction may be non-atomic: it sends messages to all other cores,
3664 performs some action on those cores, and sends messages back to the originating core. Program-order-
3665 earlier DSBs ensure that instructions program-order-earlier than the DSB are complete before sending the
3666 messages. Program-order-later DSBs ensure that all instructions program-order-after the DSB wait for those
3667 messages to return before executing.

3668 By ensuring that between any TLBI and potential same-thread access is a DSB ensures that the TLBI’s
3669 effect happens entirely between the execution of those instructions. This, coupled with the fact that these
3670 messages strengthen and never weaken the behaviour of other cores, means that you cannot observe a
3671 partial TLBI effect, as long as the programmer takes care to maintain the required thread-local ordering.
3672 This allows us to simplify the model, treating TLBIs as having a single serialisation point at which the
3673 invalidation effect happens globally.

3674 **8.6.4 Virtualization**

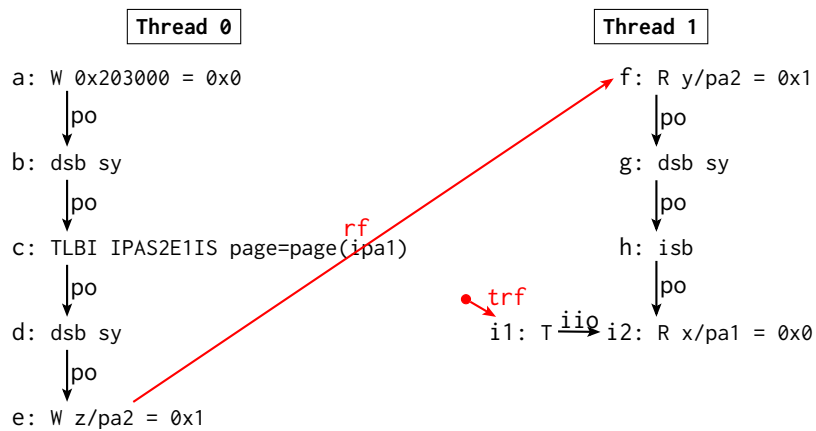
3675 Throughout this section, we have considered tests for a single-stage translation with virtual mappings.
3676 However, many of these questions and behaviours also apply to the second-stage of a two-stage mapping
3677 with intermediate physical addresses, with only a few differences. We now explore how adding a second
3678 stage of translation affects the behaviours discussed here.

3679 **Virtual to physical and IPA caches** The existence of TLBs that cache virtual to physical mappings
3680 (§8.5.4) complicates TLB maintenance requirements for changes to the intermediate physical mappings.

3681 When invalidating stale second-stage entries from the TLB, it is required for the programmer to do *two*
3682 sets of invalidations: first to invalidate any of the old cached IPA to PA entries; then, perhaps surprisingly,
3683 a second invalidation to remove any stale cached end-to-end translations, comprising whole VA to PA
3684 mappings (or combinations), as these could have indirectly cached the result of a second stage translation,
3685 without remembering the IPA they went through.

3686 This is illustrated in [MP.RT.EL2+dsb-tlbiipais-dsb+dsb-isb](#) (Figure 8.41, p.146), where invalidation of
3687 *just* the IPA is not enough to forbid the relaxed behaviour. Adding an invalidation of the VA (or all
3688 VAs), like in [MP.RT.EL2+dsb-tlbiipais-dsb-tlbiis-dsb+dsb-isb](#) (Figure 8.42, p.147), ensures that later
3689 translations cannot see the stale value any more. Note that the invalidations must happen in the specified
3690 order, as otherwise the TLB could be immediately refilled from the earlier cached second-stage entries.

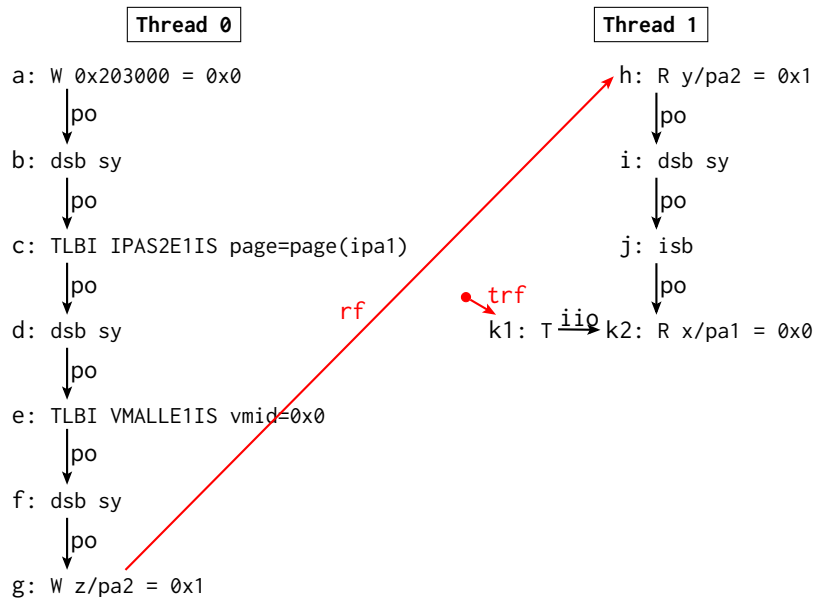
MP.RT.EL2+dsb-tlbiipais-dsb+dsb-isb		AArch64
Initial state: intermediate ipa1 ipa2, x -> ipa1, ipa1 -> pa1, y -> ipa2, ipa2 -> pa2, z -> pa2, *pa1 = 0, *pa2 = 0, 0:X0=0, 0:X1=pte3(ipa1, s2_page_table_base), 0:X2=1, 0:X3=z, 0:X4=page(ipa1), 0:PSTATE.EL=2, 1:X1=y, 1:X3=x		
Thread 0	Thread 1	Thread1 EL2 Handler
STR X0, [X1] DSB SY TLBI IPAS2E1IS, X4 DSB SY STR X2, [X3]	LDR X0, [X1] DSB SY ISB MOV X2, #1 LDR X2, [X3]	MRS X13, ELR_EL2 ADD X13, X13, #4 MSR ELR_EL2, X13 ERET
Forbidden if ETS1:X0=1 & 1:X2=0		



Despite the TLB invalidation of the stale IPA (c), a later stage 2 translation-read of that IPA (i1) can still see the old stale value.

Figure 8.41: Test MP.RT.EL2+dsb-tlbiipais-dsb+dsb-isb

MP.RT.EL2+dsb-tlbiipais-dsb-tlbiis-dsb+dsb-isb		AArch64
Initial state: intermediate ipa1 ipa2, x -> ipa1, ipa1 -> pa1, y -> ipa2, ipa2 -> pa2, z -> pa2, *pa1 = 0, *pa2 = 0, 0:X0=0, 0:X1=pte3(ipa1, s2_page_table_base), 0:X2=1, 0:X3=z, 0:X4=page(ipa1), 0:PSTATE.EL=2, 1:X1=y, 1:X3=x		
Thread 0	Thread 1	Thread1 EL2 Handler
STR X0, [X1] DSB SY TLBI IPAS2E1IS, X4 DSB SY TLBI VMALLE1IS DSB SY STR X2, [X3]	LDR X0, [X1] DSB SY isb LDR X2, [X3]	MOV X2, #1 MRS X13, ELR_EL2 ADD X13, X13, #4 MSR ELR_EL2, X13 ERET
Forbidden: 1:X0=1 & 1:X2=0		



By performing TLB invalidation of the stage 1 entries (e) after invalidating the stage 2 ones (c1), it is guaranteed that the later translation-read (k1) cannot see the old stale value any more.

Figure 8.42: Test MP.RT.EL2+dsb-tlbiipais-dsb-tlbiis-dsb+dsb-isb

3691 8.6.5 Break-before-make

3692 TLBs are not required to store only a single cached translation for a given address. There may, in
3693 general, be multiple valid translations cached in the TLB. In some cases this is perfectly fine, e.g. for
3694 translations which differ only in the permissions. However, if the translations conflict (see ‘[Conflicting](#)
3695 [entries](#)’) then having those translations both in the TLB simultaneously would be dangerous and may lead
3696 to unpredictable behaviour (see ‘[Violating break-before-make](#)’). To avoid this possibility, the architecture
3697 provides a *break-before-make* sequence, which will ensure that there cannot be two cached conflicting
3698 translations existing in the TLB at the same time.

3699 **Conflicting entries** Simply having multiple entries cached at once is not, necessarily, dangerous. Instead,
3700 the architecture specifies when such combinations are *conflicting*: when writing to the translation tables
3701 to update an already valid entry with a new valid entry, and the change involves any of the following¹:

- 3702 ▷ A change in output address, if the new or old entry is writeable.
- 3703 ▷ A change in output address, if the new and old locations have different contents.
- 3704 ▷ A change in memory type.
- 3705 ▷ A change in cacheability or shareability.
- 3706 ▷ A change in block size (e.g. replacing a page of 4KiB leaf with a 2MiB block mapping).

3707 **The break-before-make process** When updating a translation table entry to a new conflicting entry,
3708 the architecture specifies break-before-make is required, and the programmer must:

- 3709 (1) write an invalid entry to overwrite the currently valid translation table entry in memory (*break* the
3710 translation);
- 3711 (2) perform any TLB maintenance required to sufficiently invalidate the old entry from any TLB(s)
3712 required;
- 3713 (3) write the new valid translation table entry, overwriting the old invalid entry (*make* the new
3714 translation).

3715 This sequence must be well-ordered: each step must have been fully completed before the next; in
3716 particular, the write of invalid must be visible to the MMU before performing TLB maintenance, and
3717 that TLB maintenance must have completed before making the new entry. In practice, this means the
3718 sequence requires DSB SY barriers (or equivalent) before and after any TLBI instructions.

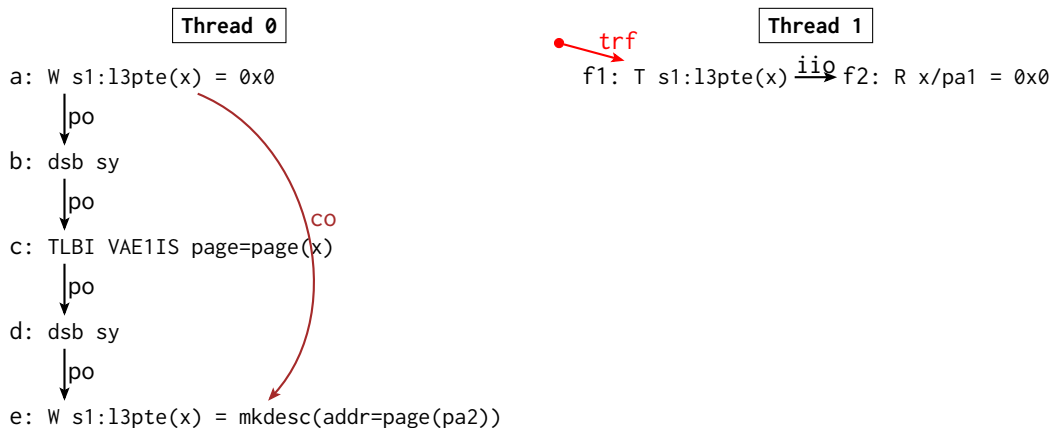
3719 **Litmus test** For completeness, the [BBM+dsb-tlbiis-dsb](#) (Figure 8.43, p.149) gives a simple valid-to-valid
3720 concurrent update test. The point is not whether a particular relaxed outcome is allowed, but that the
3721 test does not give rise to unpredictable behaviour.

3722 **Violating break-before-make** Architecturally, reaching a state where there is a pair of conflicting entries
3723 in the TLB leads to a degraded state, defined by `CONSTRAINEDUNPREDICTABLE` behaviour. The only
3724 way to avoid this is to use the appropriate break-before-make sequence. The Arm reference manual states
3725 that failure to perform break-before-make, when it is required, can lead to failure of single-copy atomicity,
3726 coherence, or even the full breakdown of uniprocessor semantics. While the reference manual does not
3727 give motivation for this, we can speculate that this is to allow hardware to perform multiple translations
3728 during execution of the instruction, for example, during hazard checking.

3729 In this work we do not try to give a characterisation of the `CONSTRAINEDUNPREDICTABLE` behaviour
3730 arising from TLB conflicts. Understanding unpredictable behaviours in full is left to future work, but a
3731 quick summary might be ‘any behaviour that an arbitrary program could have performed’. That is, an
3732 instantaneous change in the state to a random new state that would have been reachable by executing
3733 arbitrary code at that same exception level, security state, and translation regime.

¹See the Arm ARM ‘TLB maintenance requirements and the TLB maintenance instructions’ [12, D5.10.1 (p4913)] for the full list of conditions.

BBM+dsb-tlbiis-dsb		AArch64
Initial state: x -> pa1, *pa2 = 2, 0:X0=0, 0:X1=pte3(x), 0:X2=mkdesc3(oa=pa2), 0:X4=1, 0:X6=page(x), 0:PSTATE.EL=1, 1:X1=x		
Thread 0	Thread 1	Thread1 E11 Handler
STR X0, [X1] DSB SY TLBI VAE1IS, X6 DSB SY STR X2, [X1]	LDR X0, [X1]	MOV X0, #1 MRS X13, ELR_EL1 ADD X13, X13, #4 MSR ELR_EL1, X13 ERET
Allowed: 1:X0=0		



The update of the translation table entry for x in Thread 0 follows the break-before-make sequence, first *breaking* x (a), then performing the necessary TLBI sequence (b-c-d), before *making* a new mapping for x (e). This ensures the concurrent access in Thread 1 is guaranteed to see either the old value, the intermediate broken page (and so a page fault), or the new value. This test is the variant whose final state asserts that the old value was read.

Figure 8.43: Test BBM+dsb-tlbiis-dsb

8.6.6 Access permissions

Accesses which result in permission faults can have been satisfied from the TLB, and writes which update translation table entries AP field can be cached in the TLB.

Translations can give rise to permission faults. These are unlike translation faults, in that they are based not just upon the descriptor read, but also on the *kind* of access requested: read, write, or execute.

Accesses which result in permission faults result in exceptions, much like translation faults do, but, unlike those, may have been read from the TLB. This can clearly be seen in the [CoWinvTp.ro+dsb-isb](#) test (Figure 8.44, p.151), where a permission fault comes from an entry that must have read from the TLB.

Multiple cached entries We can observe multiple cached entries within a TLB by modifying the access permissions of an entry. It is not architecturally required to perform break-before-make when the two entries differ only in permissions, and it is permitted for the TLB to cache them both.

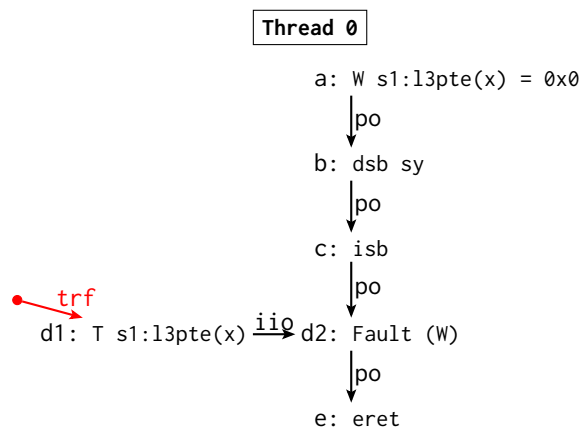
When reading from the TLB where there existing multiple entries for the same input address, it is allowed for the hardware to generate a *TLB conflict abort*.

If the hardware does not generate a conflict abort, then translation reads of that address are `CONSTRAINEDUNPREDICTABLE` as described earlier. However, when there is no requirement for break-before-make, the constraints are tighter: translations are nondeterministically able to read one or the other, (or an ‘amalgamation’) of the values [12, K1.2.3 (p11243)].

We can avoid the question of ‘amalgamation’ by constructing a test that only changes a single bit of the descriptor, in a way that is not a break-before-make violation, and therefore avoiding any questions about what amalgamations of entries are allowed. This can be seen with the [MP.RTpT.ro+dmb-dmb+dsb-isb-dsb-isb](#) test (Figure 8.45, p.152), where the existence of multiple cached entries in the TLB allows multiple translation-reads to read from different stale writes.

Atomic TLB reads The presence of multiple cached translation table entries in the TLB introduces the question of whether those TLB fills and subsequent TLB reads must read from entire single-copy atomic writes of the original translation table entries (much like a read of memory would) or whether a translation read can read from a mix of different writes. [RMD+dmb](#) (Figure 8.46, p.153) (‘Read-mixed-descriptor’) shows that translation reads cannot partially read from a write: they must read from the entire write or none of it.

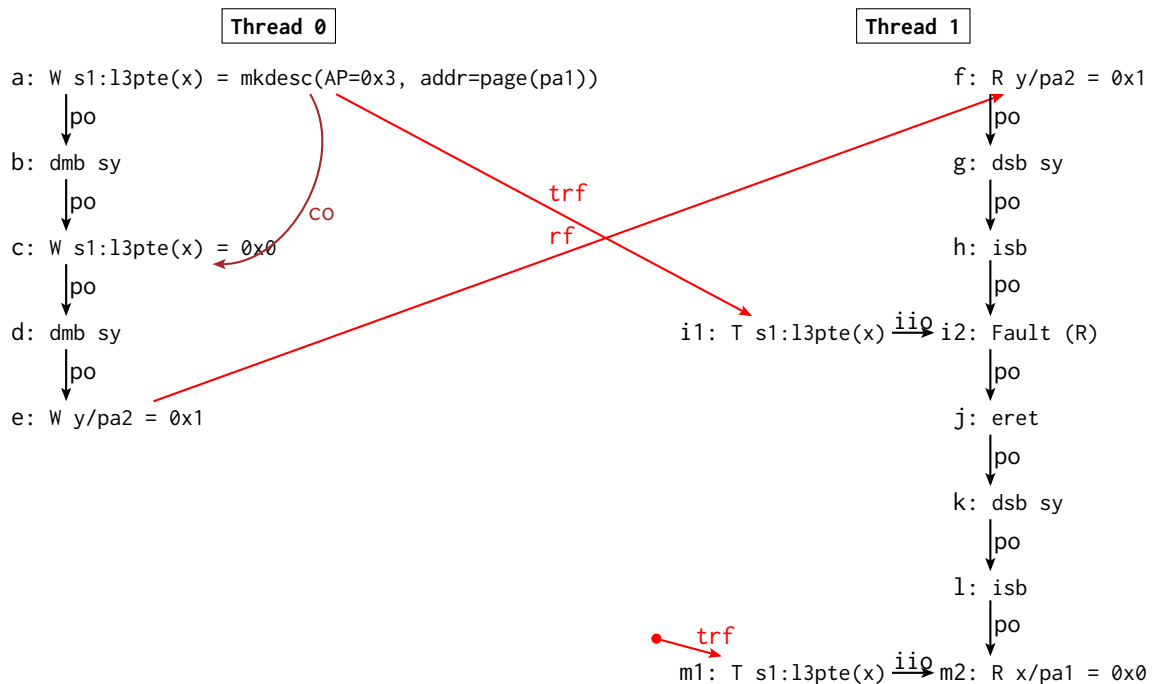
CoWinvTp.ro+dsb-isb		AArch64
Initial state: x -> pa1 with [AP = 3] and default, *pa1 = 0, 0:X0=0, 0:X1=pte3(x), 0:X2=1, 0:X3=x		
Thread 0	Thread0 EL1 Handler	
STR X0, [X1] DSB SY ISB MOV X13, #0 STR X2, [X3]	// read ESR_EL1.ISS to see if Permission or Translation fault MRS X14, ESR_EL1 AND X14, X14, #0b1111 CMP X14, #0b1111 MOV X15, #1 // Permission MOV X16, #2 // Translation CSEL X13, X15, X16, eq MRS X20, ELR_EL1 ADD X20, X20, #4 MSR ELR_EL1, X20 ERET	
Allowed: 0:X13=1		



The translation-read (d1) of x, which happens after the program-order-earlier write to the translation tables (a) because of the intervening dsb; isb sequence (b-c), can read from a stale value and result in a permission fault, as the read-only entry from the initial state may be cached in the TLB.

Figure 8.44: Test CoWinvTp.ro+dsb-isb

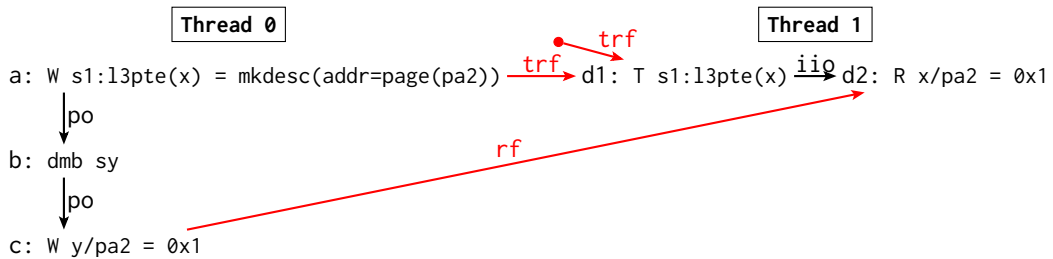
Initial state: x -> pa1 with [AP = 3] and default, y -> pa2, *pa1 = 0, 0:X0=mkdesc3(oa=pa1, AP=2), 0:X1=pte3(x), 0:X2=0, 0:X3=pte3(x), 0:X4=1, 0:X5=y, 1:X1=y, 1:X4=x		
Thread 0	Thread 1	Thread1 El1 Handler
STR X0, [X1] DMB SY STR X2, [X3] DMB SY STR X4, [X5]	LDR X0, [X1] DSB SY ISB LDR X13, [X4] MOV X2, X13 DSB SY ISB LDR X13, [X4] MOV X3, X13	// read ESR_EL1.ISS to see if Permission or Translation fault MRS X14, ESR_EL1 AND X14, X14, #0b1111 CMP X14, #0b1111 MOV X15, #1 // Permission MOV X16, #2 // Translation CSEL X13, X15, X16, eq MRS X20, ELR_EL1 ADD X20, X20, #4 MSR ELR_EL1, X20 ERET
Allowed: 1:X0=1 & 1:X2=1 & 1:X3=0		



The first translation-read of x (i1) reads from the write that removes read permissions (a) and this write must have come from the TLB because of the intervening invalidation (c), message pass (e-f), and dsb; isb sequence (g-h). The later translation-read of x (m1) can still see an even older value with read permissions, from the initial state, as it may *also* have been cached in the TLB.

Figure 8.45: Test MP.RTpT.ro+dmb-dmb+dsb-isb-dsb-isb

RMD+dmb		AArch64
Initial state: x -> pa1 with [AP = 3] and default, y -> pa2, *pa1 = 0, *pa2 = 1, 0:X0=mkdesc3(oa=pa2, AP=2), 0:X1=pte3(x), 0:X2=1, 0:X3=y, 1:X1=x		
Thread 0	Thread 1	Thread1 Handler
STR X0, [X1] DMB SY STR X2, [X3]	MOV X0, #0 LDR X0, [X1]	MRS X20, ELR_EL1 ADD X20, X20, #4 MSR ELR_EL1, X20 ERET
Forbidden: 1:X0=1		



The translation-read of x (d1) cannot read from both the 64-bit single-copy atomic write ‘a’ and the initial state. Note that this test does not, as far as we can see, violate the break-before-make requirements, as currently prescribed by the Arm manual, as the contents in memory of both locations pa1 and pa2 are the same at the time of the write to the translation tables. *isla-axiomatic* cannot generate such candidates, so the execution diagram shown is hand-written.

Figure 8.46: Test RMD+dmb

8.7 Context synchronisation

There are many operations which change the current system context. We focus on two of these: taking and returning from exceptions, and writing to system registers.

These actions can change the context that the system is executing in: the current exception level, the translation regime, the translation table base, the ASID or VMID, and the variety of other system configuration state used by the translation table walker.

8.7.1 Relaxed system registers

So far, in this and previous work, register reads and writes have been completely sequential: instructions program-order-after a write to a register always reads from that write (or an intervening write). System registers break this guarantee.

Arm System registers may require the programmer to insert explicit synchronization, as stated in the Arm reference manual [12, D13.1.2 (p5235)]:

Reads of the System registers can occur out of order with respect to earlier instructions executed on the same PE, provided that both:

- ▷ Any data dependencies between the instructions, including read-after-read dependencies, are respected.
- ▷ The reads to the register do not occur earlier than the most recent Context synchronization event to its architectural position in the instruction stream.

This means a read of a system register might not read from the most recent write to that system register.

To ensure that writes to system registers are seen by program-order-later reads, the programmer must ensure a *Context synchronization* event occurs. Context synchronisation ensures that future instructions are guaranteed to see updates to the context made by program-order-earlier instructions. Microarchitecturally,

3779 these can be achieved e.g. by flushing the pipeline, causing future instructions to restart. Some context
3780 synchronising operations have already been encountered: the ISB instruction, and taking and returning
3781 from exceptions.

3782 There are two important considerations: (1) this does not apply to non-System registers, such as the
3783 ‘Special-purpose’ registers, which never require synchronization; and (2) the synchronization required for
3784 System registers depends on the *kind* of access.

3785 There are two kinds of accesses to System registers: direct and indirect. Direct accesses are the typical
3786 way programmers interact with registers: instructions which explicitly refer to the name in its mnemonic.
3787 Indirect accesses happen when an instruction which does not explicitly mention the register by name
3788 nevertheless performs an access to it, implicitly during its execution.

3789 Out-of-order execution means these indirect register reads and writes may occur out-of-order with respect
3790 to any program-order-earlier direct reads or writes of that register. This means that before any direct
3791 read, and after any direct write, the programmer must perform a context-synchronizing event to ensure
3792 that these direct accesses occur in-order with respect to other indirect accesses. The programmer does
3793 not have to insert context-synchronization *after* any direct read, as it is guaranteed that register reads or
3794 writes cannot be affected by program-order later accesses.

3795 **System register ASL** A naive interpretation of the relaxed semantics is to allow these reads to read-
3796 from the most recent indirect write and any program-order-earlier direct writes since the last context
3797 synchronization event.

3798 However, this does not give the correct behaviour. The Arm ASL was not written in a way to accommodate
3799 relaxed system register behaviours: sometimes it re-reads the same system register multiple times, or it
3800 gets all the fields of a register in one read, or it re-uses the same previously read system register value
3801 in multiple places. This leaves open questions about whether these registers can be redundantly re-read
3802 during execution, whether the instruction reads the entire register at once or piecemeal over the course of
3803 execution, and whether repeated accesses to the same register within an instruction are able to read-from
3804 different writes. These questions, and others, are still under discussion with Arm.

3805 The model we present in the next chapter gives a simple, incomplete and possibly unsound, semantics of
3806 system registers with respect to a *pointed set* of writes (see §9.1) which allows the model to permit some
3807 of the known behaviours in this area, without yet fully exploring the architecture.

3808 **Caching of system registers in TLBs** In addition to being out-of-order due to pipeline effects, some
3809 system registers may be indirectly cached within the TLB.

3810 We have already seen one such TLB-cacheable register: the MAIR register. Direct writes to the MAIR may
3811 fail to be seen by program-order-later translations, even after context synchronization, as the translations
3812 may get their value from a stale value in the TLB which was computed using the old MAIR. To ensure
3813 that an update to a TLB-cacheable register is observed by program-order-later translations, both TLB
3814 maintenance and context synchronization are required, in that order.

3815 The registers which can be cached in this way, and the behaviours that arise from this caching, are
3816 currently under investigation with Arm.

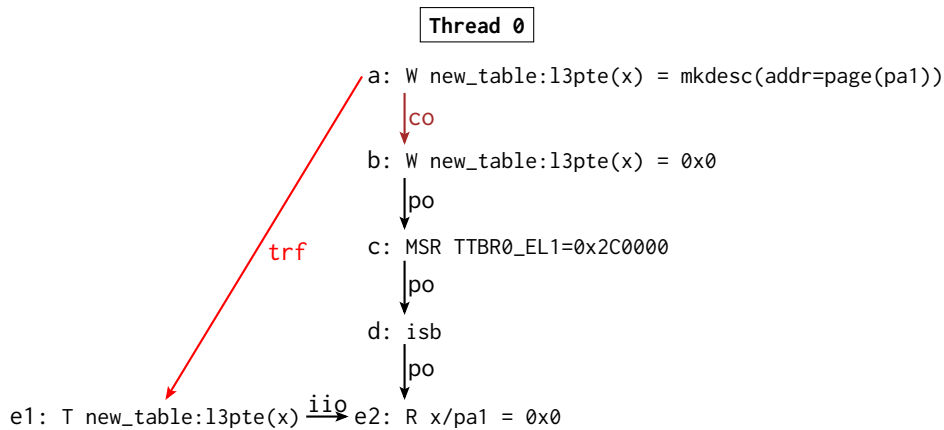
3817 **8.8 Problems**

3818 *This section describes some in-progress work with Thibaut Pérami.*

3819 Some questions, and problems, have arisen after publication of the model in the next section. These fall
3820 into two main categories:

- 3821 1. when a memory location should be considered a pagetable entry by the model ([Reachability](#)); and
- 3822 2. invalidations of block or table entries ([Wide invalidations](#)).

RUE+isb		AArch64
Initial state: *pa1 = 0, s1table new_table 0x2C0000 {, x -> invalid, }, 0:X0=mkdesc3(oa=pa1), 0:X1=0, 0:X2=pte3(x, new_table), 0:X3=ttbr(asid=1, base=new_table), 0:X4=x, 0:PSTATE.EL=1, 0:PSTATE.SP=1		
Thread 0	Thread1 El1 Handler	
STR X0, [X2] STR X1, [X2] MSR TTBR0_EL1, X3 ISB MOV X1, #1 LDR X3, [X4]	MRS X13, ELR_EL1 ADD X13, X13, #4 MSR ELR_EL1, X13 ERET	
Final state: 0:X1=1		



The write to the `new_table` translation table entry for `x` (a) is not visible at the point of the change of TTBR (c), and so the later translation table walk (e1) cannot read from it. Note that `isla-axiomatic` currently does not do any kind of reachability analysis, and so does not forbid this test.

Figure 8.47: Test RUE+isb

8.8.1 Reachability

One important property that the TLB must have is that it may only add new cached translations for translation table entries which are *reachable* by a translation in the current context. That is, it can only cache an entry which is the result of a valid translation table walk, either using values from memory or other valid translation table entries from the TLB, using the current translation table base and other System register state. This means that writes which are coherence-before the most recent write, at the time a translation table entry location becomes reachable, are not visible to the walker, and therefore cannot have been cached in any TLB.

This is captured in the `RUE+isb` (Figure 8.47) (‘Read-unreachable-entry’) test, which is architecturally forbidden as the write to the translation table from before the time the location becomes reachable by translation table walkers cannot have been cached in any TLBs, or read from by any spontaneous walks. Currently, the models do not attempt to track reachability, and so erroneously allow this test.

8.8.2 Wide invalidations

In §8.6, we discussed invalidations of entries in the TLB, and investigated how TLBI instructions remove cached translations which translate a given page. This raises an important question: does the invalidation apply only to that page, or to all translations mapped by the same translation table entry? On one hand, TLBs can split such ‘wide’ translations into multiple smaller paged-sized ones, e.g. for when the stage 2 mapping is at a smaller granularity. On the other hand, this would require software to do a very expensive

InvalidateWideBlock		AArch64
Initial state: aligned 2097152 virtual x, x -> pa1 at level 2, 0:X1=pte2(x,page_table_base), 0:X3=x, 0:X4=page(x), 0:PSTATE.EL=1, 0:PSTATE.SP=1		
Thread 0	Thread0 EL1 Handler	
MOV X2, #0 STR X2, [X1] DSB SY TLBI VAE1, X4 DSB SY ISB LDR X6, [X3, #0x1000]	MOV X6,#1 MRS X13,ELR_EL1 ADD X13,X13,#4 MSR ELR_EL1,X13 ERET	
Allowed: 0:X6 = 0		

x is mapped by a 2 MiB block entry at level 2. Breaking it and invalidating the TLB passing x affects all translations in the same 2 MiB block.

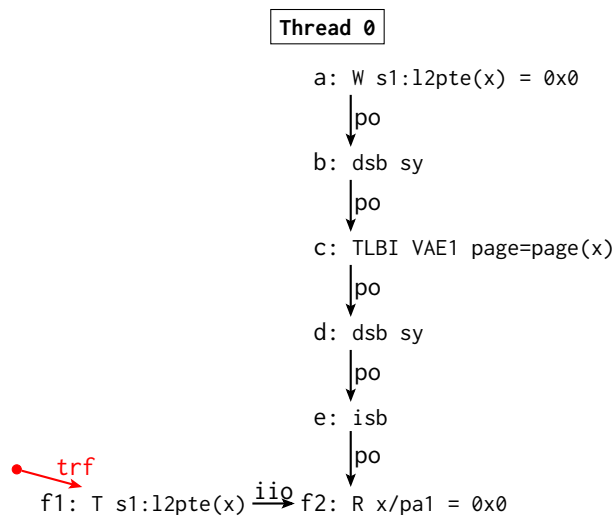


Figure 8.48: Test InvalidateWideBlock

3841 invalidation to clear cached block entries, either iterating over every page in the region, or simply flushing
3842 the entire TLB.

3843 Arm have, tentatively, decided that the architectural intent is that the TLB invalidations should invalidate
3844 all mappings which use the same cached translation table entry, see [InvalidateWideBlock](#) (Figure 8.48).
3845 However, this does not apply when the original mappings were of smaller granularity. For example, even
3846 if writing an invalid entry at level 2 then doing invalidation, the old level 3 entries may still be cached in
3847 the TLB, illustrated in [InvalidateWide](#) (Figure 8.49, p.157).

3848 8.9 Contributions

3849 We have now covered all the key relaxed virtual memory behaviours, and will in the next chapter move on
3850 to discuss the model which captures those behaviours. But before that, it may at this point be unclear
3851 what the *contribution* of this chapter is. They come in three forms: (1) the attempt at some systematic
3852 coverage of the kinds of behaviours which systems software must account for; (2) the precise, formal
3853 description (in prose, and as litmus tests) of those behaviours; and, (3) the clarification of the architecture
3854 where such behaviours were otherwise unclear.

3855 **Coverage of behaviours** While this chapter attempts to systematically cover the behaviours we imagine
3856 software may try to rely on, starting from the basics of translation table walks and exploring the effects of
3857 out-of-order pipelines, caching, and barriers, we cannot claim it is *exhaustive*. As it is a manually compiled
3858 and curated list of behaviours, there will be corner cases missed and software patterns overlooked. That
3859 said, we believe we have covered those patterns which are known for the features we cover, enough for
3860 software verification efforts of microkernels and hypervisors.

InvalidateWide		AArch64
Initial state: *pa1 = 1, *pa2 = 2, x -> pa1, z -> pa2, 0:X1=pte2(x,page_table_base), 0:X3=x, 0:X4=page(x), 0:X5=z, 0:PSTATE.EL=1, 0:PSTATE.SP=1		
Thread 0	Thread0 El1 Handler	
MOV X2, #0 STR X2, [X1] DSB SY TLBI VAE1, X4 DSB SY ISB LDR X6, [X5]	MOV X6, #0 MRS X13,ELR_EL1 ADD X13,X13,#4 MSR ELR_EL1,X13 ERET	
Allowed: 0:X6 = 2		

x and z are two entries mapped at level 3 but within the same 2 MiB region, so are mapped by the same level 2 entry. Breaking the level 2 entry and invalidating with one address does not invalidate the other.

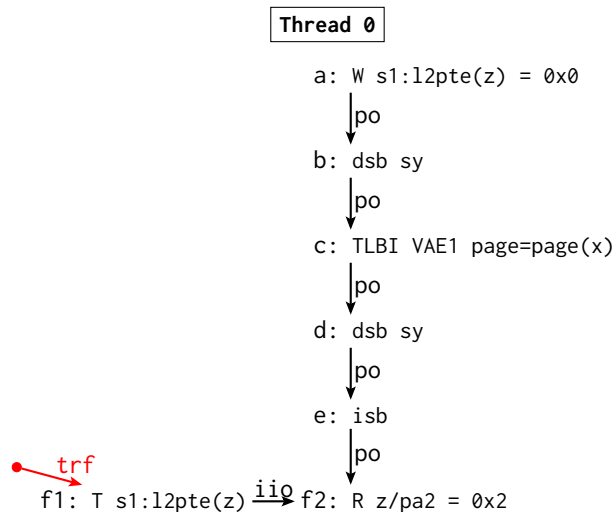


Figure 8.49: Test InvalidateWide

3861 **Clarification of architecture** Attempts to clarify the architecture come primarily from confidential
3862 discussions with architects. The behaviours discussed usually fell into one of three categories: whether
3863 they were clear already; needed further exploration; or are still under investigation by Arm.

3864 The first major category are those behaviours which were already clear and covered in the architecture
3865 text. As alluded to right at the start of this chapter, these are not whole sections or sub-sections or even
3866 necessarily whole tests. The most obvious cases are §8.3.3 ('Invalid entries'), §8.2.1 ('Virtual coherence'),
3867 and §8.6.5 ('Break-before-make'). These are fundamental behaviours to the correctness of all modern
3868 systems software, and for which the architecture reference manual has clear words (at least, enough to
3869 cover the basic sequences software rely upon).

3870 Most of the subsections fall into a more involved category, of things that either had some associated reference
3871 materials, or was otherwise clear from discussion with architects, but for which further investigation was
3872 needed. This includes: forwarding (§8.4.4) and speculation (§8.4.5) for translation table walks; multi-copy
3873 atomic translation table walks (§8.4.7); intra-instruction ordering (§8.4.8, §8.4.9); micro-TLBs (§8.5.3)
3874 and partial walk caching (§8.5.4); a variety of TLBI questions (§8.6); and, system register accesses (§8.7.1).

3875 Despite the work conducted here, from reading the architecture reference text, discussions with architects,
3876 and the testing of existing hardware, there are still many questions, some of which are currently under
3877 investigation by Arm. These include further questions about the scope of TLBIs, interaction with exceptions
3878 and interrupts, changes in cacheability, translations for instruction fetching, and relaxed system register
3879 accesses. Those areas will require more work before giving a concrete semantics.

3880 **8.10 Related work**

3881 The authoritative Arm-internal ASL model [10, 11, 72], and the Sail model derived from it [45] cover
3882 address translation, and other features sufficient to boot an OS (Linux), as do the handwritten Sail models
3883 for RISC-V (Linux and FreeBSD) and MIPS/CHERI-MIPS (FreeBSD, CheriBSD), but without any cache
3884 effects.

3885 Goel et al. [85, 98] describe an ACL2 model for much of x86 that covers address translation; and the
3886 Forvis [99] and RISC-V-PLV [100] Haskell RISC-V ISA models are also complete enough to boot Linux.

3887 Syeda and Klein [101, 102] provide a somewhat idealised model for ARMv7 address translation and TLB
3888 maintenance.

3889 Komodo [58] uses a handwritten model for a small part of ARMv7, as do Guanciale et al. [59, 60].
3890 Romanescu et al. [103, 104] discuss address translation in the concurrent setting, but with respect to
3891 idealised models.

3892 Lustig et al. [87] describe a concurrent model for address translation based on the Intel Sandy Bridge
3893 microarchitecture, combined with a synopsis of some of the relevant Linux code, but not an architectural
3894 semantics for machine-code programs. Hossain [105] continues this line, producing an estimated architec-
3895 tural model for x86, using their *TransForm* framework to automatically synthesise litmus tests from the
3896 memory model used in validation.

3897 Tao et al. [106] define a weak virtual memory model, based on what they call wDRF (weak data race
3898 freedom), which gives bounds to software when it behaves in accordance with a set of conditions, which
3899 they use in the verification of their *SeKVM* hypervisor. These conditions permit some races to the page
3900 table code, but do not attempt to capture the full architectural envelope. The model here, we hope, gives
3901 a sound underlying base from which one could prove a wDRF-SC property above, connecting the work of
3902 Tao to our model.

3903 Independent work by Arm, after the conclusion of this work, further extended the herdttools suite of
3904 tools, models, and tests, for some overlapping portion of the VMSA, including covering dirty bits and
3905 access flags which we do not consider here [107]. They differ from the model presented here where the
3906 architecture has changed (around ETS and write forwarding). Further work is required to understand
3907 how the models differ in their intersection and how a model that covers their union could be constructed.

An axiomatic VMSA model

We now define a semantic model for Arm-A relaxed virtual memory (*RVM*) that, to the best of our knowledge, captures the Arm architectural intent for the questions discussed in [Chapter 8](#), including two-stage translation-table walks and the required TLB maintenance, as an extension to the base usermode Arm-A axiomatic memory model [7] (as recalled in [Chapter 2](#)).

In [Chapter 8](#), we described the design issues in microarchitectural terms, discussing the behaviour of translation table walks and TLB caching, along with the needs of system software. We now abstract from those specific litmus-test examples and give a general definition of what behaviour is allowed or not. We do this by extending the base model, defining ordering between new translation-read events and other events, without modelling TLBs or out-of-order pipelines directly.

The base Arm axiomatic model is defined as a predicate over candidate executions, each of which is a graph with various events (reads, writes, barriers) and relations over them. We now extend these candidates with new events and new relations over those events, and modify some original relations.

9.1 Candidate events

We extend the events of the candidate executions and the corresponding labelling function, given in [Figure 9.1](#), to contain new events:

- ▷ T for the implicit reads of memory originating from architected translation-table walks. These roughly correspond to the point the translation-read was satisfaction from memory, which with TLBs may happen very early¹.
- ▷ TLBI events for each TLBI instruction, with a single such event per TLBI instruction, corresponding to the TLBI being completed on all relevant cores.
- ▷ TE and ERET events for taking and returning from an exception, annotated with the reason for the exception (not shown here).
- ▷ MSR events for writes to relevant system registers, such as the TTBRs; and MRS for reads.
- ▷ DSB events for DSB instructions.

Implicit accesses and faults Execution of the translation in the Arm architectural pseudocode performs reads of memory, which would normally generate R events in the candidate executions. Instead, when those reads happen during calls to that function, we label them as T events. This means that each translation table walk may generate up to 24 T events, before the instruction generates the R or W event. We explored alternative representations, including collecting all reads into a single large translation event, or placing all translations into the standard R set. These options have advantages, but we made the choice to keep a 1-to-1 correspondence between the events of the execution and the ISA, and to retain as much of the original 2018 model events and relations unchanged as possible.

We also choose not to include TLB hits and misses in the model directly, but instead model the TLB as a relaxation of the values the walk can read from, much like normal data memory read events and modelling load buffering, write gathering, and caches.

¹Analogously to R events and load-buffers.

$$\begin{aligned}
\text{Label} &\equiv \text{Reads} \cup \text{Writes} \cup \text{Barriers} \cup \text{Translations} \cup \text{TLBIs} \cup \text{Exceptions} \cup \text{SysRegs} \\
\text{Reads} &\equiv \{\mathbf{R}, \mathbf{A}, \mathbf{Q}\} \times \text{Loc} \times \text{Val} \\
\text{Writes} &\equiv \{\mathbf{W}, \mathbf{L}\} \times \text{Loc} \times \text{Val} \\
\text{Barriers} &\equiv \{\mathbf{DMB.LD}, \mathbf{DMB.ST}, \mathbf{DMB.SY}, \mathbf{DSB.SY}, \mathbf{ISB}\} \\
\text{Translations} &\equiv \{\mathbf{T}\} \times \text{PA} \times \text{TranslationInfo} \\
\text{TLBIs} &\equiv \{\mathbf{TLBI}\} \times \text{TLBIOp} \times \text{Shareability} \times \text{Regime} \times \text{VMID?} \times \text{ASID?} \times \text{Addr?} \\
\text{Exceptions} &\equiv \{\mathbf{TE}\} \times \text{ExceptionInfo} \cup \{\mathbf{ERET}\} \\
\text{SysRegs} &\equiv \{\mathbf{MSR}, \mathbf{MRS}\} \times \text{SysRegName} \times \text{Val} \\
\text{VA, IPA} &\equiv \text{Addr} \equiv \text{Bitvec}_{48} \\
\text{Loc} &\equiv \text{PA} \equiv \text{Bitvec}_{64} \\
\text{Val} &\equiv \text{Bitvec}_{64} \\
\text{TranslationInfo} &\equiv \text{VA} \times \text{IPA?} \times \text{Level} \times \text{Stage} \\
\text{TLBIOp} &\equiv \{\mathbf{VA}, \mathbf{IPA}, \mathbf{ALL}, \mathbf{ASID}, \mathbf{VMALL}, \dots\} \\
\text{ASID, VMID} &\equiv \text{Bitvec}_8 \\
\text{Regime} &\equiv \{\mathbf{EL1\&0}, \mathbf{EL2}\} \\
\text{Shareability} &\equiv \{\mathbf{NSH}, \mathbf{ISH}\} \\
\text{SysRegName} &\equiv \{\mathbf{TTBR0_EL1}, \mathbf{TTBR0_EL2}, \mathbf{VTTBR_EL2}, \dots\} \\
\text{ExceptionInfo} &\equiv \dots
\end{aligned}$$

where T? signifies an optional field of type T.

Figure 9.1: Definition of candidate event labels for Arm-A RVM candidates. Parts which differ from the original definition are highlighted in blue. Terminal words of the label are given in bold.

3945 We add a helper set, T_f , for translation reads which read-from a write whose ‘valid’ bit is 0. If a
3946 translation read results in a fault (either because it was an invalid entry and we get a translation fault, or
3947 because the access permissions of the resulting translation do not permit the kind of requested access and
3948 so result in a permission fault), the candidate will contain a **Fault** event (partitioned into **Fault_t** and
3949 **Fault_p** for translation and permission faults) in po order where the explicit memory event would have
3950 been. See the discussion on obETS (§9.4.6) for more explanation of these ‘ghost’ fault events.

3951 We partition the T set into two subsets: **Stage1** and **Stage2** for translation read events from a stage 1 or
3952 stage 2 walk respectively (stage 2 reads during a stage 1 walk are marked as Stage 2, not Stage 1).

3953 Finally, we leave the M set unchanged, containing only explicit reads and writes performed by instructions.

3954 **TLBIs** As described in §7.7, Arm have a variety of TLBI instructions, with varying arguments. Each
3955 TLBI instruction generates a single TLBI event, each with a distinct label. To aid in modelling, there are a
3956 set of subsets of TLBI for various kinds of TLBI:

- 3957 ▷ TLBI-S1 for invalidations of Stage 1 entries.
- 3958 ▷ TLBI-S2 for invalidations of Stage 2 entries.
- 3959 ▷ TLBI-IPA for invalidations by intermediate physical address.
- 3960 ▷ TLBI-VA for invalidations by virtual address.
- 3961 ▷ TLBI-ASID for invalidations by ASID.
- 3962 ▷ TLBI-VMID for invalidations by VMID.
- 3963 ▷ TLBI-ALL for the TLBI ALL instructions.
- 3964 ▷ TLBI-IS for broadcast TLBIs.
- 3965 ▷ TLBI-EL1 for invalidations of the EL1&0 regime.
- 3966 ▷ TLBI-EL2 for invalidations of the EL2 regime.

3967 These do not *cut* the TLBI set into partitions. Rather any TLBI event may belong to multiple. For example,
3968 a TLBI VAE1IS event would belong to TLBI-VA, TLBI-VMID, TLBI-EL1, and TLBI-IS.

3969 We also include all TLBIs in a general C (‘Cache maintenance’) set.


```

1 let dsbsy = DSBISH | DSBSY | DSBNSH
2 let dsbst = dsbsy | DSBST | DSBISHST | DSBNSHST
3 let dsbld = dsbsy | DSBLD | DSBISHLD | DSBNSHLD
4 let dsbnsh = DSBNSH
5 let dmbsy = dsbsy | DMBSY
6 let dmbst = dmbsy | dsbst | DMBST
7           | DSBST | DSBISHST | DSBNSHST
8 let dmbld = dmbsy | DMBLD
9           | dsbld | DSBISHLD | DSBNSHLD
10 let dmb = dmbsy | dmbst | dmbld
11 let dsb = dsbsy | dsbst | dsbld

```

Figure 9.2: Barrier helper sets.

3970 **Exceptions** Despite not modelling exceptions in general in this part, we do need to include some exception
3971 machinery in the model to capture the minimal ordering requirements arising from both their context
3972 synchronisation effects and behaviours from crossing exception level boundaries.

3973 To support this, we add two new events: TE (‘Take exception’); and ERET (‘Exception return’).

3974 **Barriers** The Arm DSB (‘Data synchronization barrier’) instruction is required for TLB maintenance, as
3975 was seen in the previous chapter. We include DSB events, one for each kind of DSB instruction:

- 3976 ▷ DSBSY and DSBISH (which we treat as equivalent, as we do not model shareability domains).
- 3977 ▷ DSBNSH, for non-shareable (thread-local) DSBs.
- 3978 ▷ DSBST, DSBLD, for DSBs with ST or LD kinds.
- 3979 ▷ DSBISHST, DSBISHLD, and so on, for all combinations of DSB instruction domain and access types.

3980 Arm define a hierarchy of barriers where, for example:

$$\text{DMB.LD} < \text{DMB.SY} < \text{DSB.SY}$$

3981 That is, any ordering imposed by a DMB.LD is also imposed by a DMB.SY, and therefore also a DSB.SY.

3982 To avoid an explosion in the number of relations as we add the new barrier events, we simplify and update
3983 the barrier-ordered-before relation in the Arm model to use a collection of helper sets, which encode this
3984 hierarchy. Those helper sets can be found in Figure 9.2.

3985 **Context changing and synchronisation** Finally, we add events for context-changing and context-
3986 synchronising operations. Context changes are updates to system registers which change the current
3987 translation regime, which are generated as MSR events We add a general context-synchronisation event set
3988 CSE which includes ISB, TE, and ERET.

3989 Changes to system registers may have relaxed behaviours, as described in §8.7.1, but full relaxation of
3990 the system register reads done by the Arm pseudocode is unlikely to be valid, consistent, or meaningful.
3991 Instead, we introduce a *pointed-set semantics*: when generating a candidate, we keep a per-system-register
3992 set of writes to that register, remembering which one is the most recent. On a write to that system
3993 register, we add it to the pointed set as the new pointed element. On a read of that system register, we
3994 generate one candidate for each value in the set, and then ‘lock’ the remainder of the execution of that
3995 instruction to that value, so repeated reads will see the same value. When a context-synchronization
3996 event is generated (that is, an event that will be in the CSE set) all the sets are reduced to singleton sets
3997 containing only the most recent write.

3998 This gives us some relaxed behaviours, enough to see relaxed behaviours around changes to the TTBR, but
3999 we note that this is unlikely to be the full story for relaxed system registers.

9.2 Candidate relations

We extend the set of candidate relations and the witness to include the new events: adding the explicit events to program-order, updating address dependencies to take translation into account, and adding new relations to capture intra-instruction ordering and ordering to/from translation-reads and TLBIs. The updated candidate relations are given in Figure 9.3.

Addresses, ASIDs, and VMIDs Each translation table walk will read from general-purpose and system registers to get a value for the input address, the current ASID, current VMID, and the roots of the translation tables. We then relate each T with any other T where the translation associated with it is for the same virtual address (with `same-va`), the same intermediate-physical address (with `same-ipa`), or the same resulting physical address (`same-pa`). This means that all T events within a translation have the same `same-*` relations. We also include `same-*-page` relations, which relate two events when their virtual, intermediate physical, or physical addresses, are in the same page.

If two translations are for the same ASID, their translation reads are related by `same-asid`. If two translations are for the same VMID, their translation reads are related by `same-vmid`.

To use these relations, we also include TLBI events. A TLBI-X is related to T by `same-X` if the parameter to the TLBI instruction (the page, VMID, or ASID) matches the T event's associated translation. For example, a TLBI-IPA event would be `same-ipa-page` related to a T whose translation was for an intermediate physical address in the page provided as the parameter to the TLBIIPA instruction.

Generalised coherence order We add an extended coherence order `wco`, which is an arbitrary linearisation of writes, DSB barriers, and cache and TLB maintenance operations, consistent with the usual coherence order. This generalised coherence order captures a global 'commit' order of these operations, consistent with what a hypothetical microarchitectural-style operational semantics would generate.

One might be concerned at the validity of doing this, for two reasons. First, this generalised coherence order will relate all writes, not just same-location ones. However, extending coherence to a total order over all locations is sound [6, §10.5 p174], so this does not cause an issue. Secondly, it enforces a kind of atomicity of TLBIs. For broadcast TLBIs, microarchitectures will implement these with message passing to and from each core separately, and so there is no single moment the TLBI 'happens'. However, as described in §8.6.6, we are able to consider TLBI instructions as executing 'atomically', so long as there are no break-before-make violations. This is a similar justification as for including DC and IC events in a similar generalised coherence order for instruction fetching [35, §5 p29].

The full definition of `wco`, as defined in `isla-axiomatic`, can be found in Figure 9.8, p.173.

Dependencies A candidate execution consists not only of events, and reads-from relations but also a set of dependencies: `addr`, `data`, `ctrl`, `po`, and `loc`. We add `iio`, and a special `tdata` (described below) to these.

The intra-instruction-order relation (`iio`) relates two events from the same instruction in the order the intra-instruction semantics generated the events. This relation therefore captures a total order over all events within an instruction, regardless of the intra-instruction dependencies (control, data) or unordered accesses (for example, for misaligned accesses). We are currently investigating a relaxation of this ordering, and associated changes in the underlying Arm pseudocode definitions, to enable a more relaxed definition of the ordering within an instruction to handle these cases.

We make `loc` relate events with the same physical address (for T events, this is the physical location of the translation table entry).

Program order (`po`) is restricted to explicit events: R, W, F, C, CSE and MSR. Implicit translation reads (T) and any indirect reads or writes of registers are not included in `po`.

Address dependencies were once fundamental, with address translation we can now define address dependencies as dependencies into the translation table walk. To do this, we include a new relation, `tdata`, that relates reads with the translation read events of any translation which reads from the register written by that read to compute the address. The traditional `addr` can then be recovered as `tdata; iio*; [M]`.

The Arm-A RVM pre-execution relations are:

- ▷ **intra-instruction-order**: E_1 **iiio** E_2 for events E_1, E_2 in the same instruction where E_1 is generated before E_2 in the intra-instruction trace.
- ▷ **program order**: E_1 **po** E_2 for **explicit events** E_1, E_2 such that the instruction generating E_1 occurs before the instruction generating E_2 in the instruction stream.
- ▷ **same-location**: M_1 **loc** M_2 iff the address of M_1 is the same **physical** location as used by M_2 .
- ▷ **same-address (same-va, same-ipa, same-pa)**: between events E_1, E_2 iff the (virtual/intermediate physical/physical)-address of E_1 is the same as E_2 .
- ▷ **same-page (same-va-page, same-ipa-page, same-pa-page)**: between events E_1, E_2 iff the (virtual/intermediate physical/physical) address of E_1 is in the same page (e.g. 4KiB chunk) as E_2 .
- ▷ **same-address-space (same-asid, same-vmid)**: between events E_1, E_2 iff the associated translation of E_1 and E_2 are using the same ASID or VMID.
- ▷ **address dependent**: R_1 **tdata** T_2 iff the value read by R_1 is used in the calculation of the address which T_2 is a translation of.
- ▷ **data dependent**: R_1 **data** W_2 iff the value read by R_1 is used in the calculation of the value written by W_2 .
- ▷ **control dependent**: R_1 **ctrl** E_2 iff the value read by R_1 is used to determine whether or not the instruction E_2 originates from would have executed at all.
- ▷ **read-modify-write**: R_1 **rmw** W_2 for the separate read and write events of an atomic update.
- ▷ **external**: E_1 **ext** E_2 iff the instructions which generated events E_1 and E_2 originated from different hardware threads.

Plus the existentially quantified witness:

- ▷ **reads-from (rf)**, from W_1 to R_2 when R_2 reads the value that W_1 wrote.
- ▷ **translation-reads-from (trf)**, from W_1 to T_2 when T_2 reads the value that W_1 wrote.
- ▷ **coherence-order (co)**, from W_1 to W_2 where W_1 appears before W_2 in the coherence order of that location, (informally, that W_1 propagated to memory before W_2).

where E_n represents events of any kind, M_n is an **explicit** memory effect event, T_n is a translation-read event, R_n is a read event, and W_n is a write event.

Figure 9.3: Definition of the candidate relations and witness for Arm-A RVM candidates. Parts which differ from the base definition are highlighted in blue.

9.3 Axioms

The RVM model axioms are, mostly, a syntactic extension to the original Arm-A axiomatic model presented in Chapter 2. This is by design. Although there may be other nicer or more succinct ways of phrasing the model, the variation presented here is designed to be as syntactically close as possible to the original. This helps with readability for those familiar with the original; it allows us to present the differences to the original in an easier form; it makes recovery of the original model easier; and, it makes it easier to prove equivalence of the axiomatic models in the presence of constant address translation, increasing the confidence we have in the model.

Figure 9.4 contains the axioms and relations for our Arm-A relaxed virtual memory axiomatic model, defined in Cat. Unchanged parts from the original are greyed out. We elide some helper relations, which we will describe in more detail later.

The model has three kinds of axioms: internal ones for per-location guarantees, an external axiom for the global happens-before ordering, and the atomic axiom for RMWs (untouched in this work).

Internal axioms The new model has two per-location axioms: `internal` and `translation-internal`.

```
1 (* Internal visibility requirement *)
2 acyclic po-loc | fr | co | rf as internal
3
4 (* Writes cannot forward to po-future translates *)
5 acyclic (po-pa | trfi) as translation-internal
```

Unchanged from the original, the `internal` axiom captures the SC-per-location guarantee. Translations, however, do not have the same per-location guarantees. To account for this, we introduce a second axiom, `translation-internal`, which captures the weaker per-location guarantee for translation table walks. Since translation reads, in the presence of TLB caching and out-of-order pipelines, do not even guarantee coherence, the only behaviour that this axiom ends up preventing is translation reads reading from program-order later stores.

External axiom The external axiom asserts acyclicity of the global happens-before ordering for Arm. The happens-before relation (called `ob`, ‘ordered-before’, in Arm) is the union of all the ordering relations, given in §9.4.

```
1 (* Ordered-before *)
2 let ob = (obs | dob | aob | bob | iio | tob | obtlbi | ctxob
           | obfault | obETS)+
3
4 (* External visibility requirement *)
5 irreflexive ob as external
```

We choose to include all the pipeline and TLB effects as ordering requirements, rather than introducing new ordering axioms just for translation and TLB invalidation. This produces a model that is more consistent with the previous Arm memory models, and ensures ordering information gained through observing translation table walks is respected by non-translation-table accesses.

Atomic axiom The atomic axiom remains unchanged. In this work, we do not consider the interaction of translation with atomic accesses, although one expects the intra-instruction semantics defined by the ASL already describes the behaviour in enough detail.

```
1 (* Atomic requirement *)
2 empty rmw & (fre; coe) as atomic
```

```

1  let speculative =
2    ctrl
3  | addr; po
4  | [T]; instruction-order
5
6  (* translation-ordered-before *)
7  let tob =
8    [T_f]; tfr
9  | [T]; iio; [R|W]; po; [W]
10 | speculative; trfi
11
12 (* observed by *)
13 let obs =
14   rfe | fr | wco
15 | trfe
16
17 (* ordered-before TLBI and translate *)
18 let obtlbi_translate =
19   [T&Stage1]; tlb_barriered; [TLBI-S1]
20 | ([T&Stage2]; tlb_barriered; [TLBI-S2])
21 &
22   (same-translation; [T&Stage1]
23    ; trf-1; wco-1)
24 | ([T&Stage2]; tlb_barriered; [TLBI-S2]
25    ; wco?; [TLBI-S1])
26 &
27   (same-translation; [T&Stage1]
28    ; maybe_TLB_cached)
29
30 (* ordered-before TLBI *)
31 let obtlbi =
32   obtlbi_translate
33 | [R|W|Fault_T]; iio-1; [T]
34   ; (obtlbi_translate & ext); [TLBI]
35
36 (* context-change ordered-before *)
37 let ctxob =
38   speculative; [MSR]
39 | [CSE]; instruction-order
40 | [ContextChange]; po; [CSE]
41 | speculative; [CSE]
42 | po; [ERET]; instruction-order; [T]
43
44 (* ordered-before a fault *)
45 let obfault =
46   data; [FaultFromW]
47 | speculative; [FaultFromW]
48 | [dmbst]; po; [FaultFromW]
49 | [dmbld]; po; [FaultFromW|FaultFromR]
50 | [A|Q]; po; [FaultFromW|FaultFromR]
51 | [R|W]; po; [FaultFromReleaseW]
52
53 (* ETS-ordered-before *)
54 let obETS =
55   (obfault; [Fault_T]); iio-1; [T_f]
56 | ([TLBI]; po; [dsb]
57    ; instruction-order; [T])
58   & tlb-affects
59
60 (* dependency-ordered-before *)
61 let dob =
62   addr | data
63 | speculative; [W]
64 | addr; po; [W]
65 | (addr | data); rfi
66 | (addr | data); trfi
67
68 (* atomic-ordered-before *)
69 let aob =
70   rmw
71 | [range(rmw)]; rfi; [A|Q]
72
73 (* barrier-ordered-before *)
74 let bob =
75   [R]; po; [dmbld]
76 | [W]; po; [dmbst]
77 | [dmbst]; po; [W]
78 | [dmbld]; po; [R|W]
79 | [L]; po; [A]
80 | [A|Q]; po; [R|W]
81 | [R|W]; po; [L]
82 | [F|C]; po; [dsbsy]
83 | [dsb]; po
84
85 (* Ordered-before *)
86 let ob =
87   (obs | dob | aob | bob
88    | iio | tob | ctxob
89    | obtlbi | obfault | obETS)+
90
91 (* Internal visibility requirement *)
92 acyclic po-loc | fr | co | rf as internal
93 (* External visibility requirement *)
94 irreflexive ob as external
95 (* Atomic requirement *)
96 empty rmw & (fre; coe) as atomic
97 (* Writes cannot forward to po-future
98    translates *)
98 acyclic (po-pa | trfi)
99   as translation-internal

```

Figure 9.4: RVM axioms and relations

9.4 Relations

The RVM model modifies some of the original ordering relations, and introduces some new ones. This section goes through each in detail, describing the mechanisms, and justifying the existence or non-existence of particular clauses.

9.4.1 Observed-by

```
1 (* observed by *)
2 let obs = rfe | fr | wco | trfe
```

The ‘observed-by’ relation includes the original `rf` and `fr` (over physical locations), the ‘*generalised coherence order*’ (`wco`, §9.2), and the translation-reads-from-external (`trfe`) relation.

Generalised coherence Including `wco`, which is existentially quantified over the candidates, fixes some global order in which the writes and TLBIs happen in. Consider, informally, some microarchitectural execution. Writes would be propagated to the coherent storage subsystem, and TLBIs would be completed, and these would be interleaved within some global-time whole-machine trace. The generalised `wco` relation captures a serialisation of these events. The model is then quantified over all such orderings, accounting for any interleaving of these events.

External translation reads Inclusion of `trfe` enforces that those translation-table-walk translation reads which could not come from forwarding must have originally come from the coherent storage subsystem, and so the write must have been globally propagated before the translation read happened (§8.4.2, §8.4.7).

However, the translation read might have happened much later, either due to out-of-order execution (§8.4.1) or TLB caching (§8.5.1). So we do not include the translation analogue of the usual from-reads-external relation in `ob`, `tfre` (translation-from-reads-external), which relates translation-reads with writes coherence-before the ones they read from.

Additionally, writes may be forwarded to its thread’s translation walker: translation-reads may be satisfied before the propagation of the write to the coherent storage subsystem (§8.4.4). Therefore we do not include the translation analogue of reads-from-internal in `ob`, `trfi` (translation-reads-from-internal), which relates writes with same-thread translation-reads which read from that write.

9.4.2 Dependency-ordered-before

```
1 let dob =
2   addr | data
3   | speculative; [W]
4   | addr; po; [W]
5   | (addr | data); rfi
6   | (addr | data); trfi
```

The dependency-ordered-before relation is mostly unchanged: we add a single `(addr | data); trfi` clause to forbid thin-air creation of values (§8.4.1, §8.4.2), much like the analogous constraint in the usermode model.

9.4.3 Barrier-ordered-before

```
1 let bob =
2   [R]; po; [dmbld]
3 | [W]; po; [dmbst]
4 | [dmbst]; po; [W]
5 | [dmbld]; po; [R|W]
6 | [L]; po; [A]
7 | [A | Q]; po; [R | W]
8 | [R | W]; po; [L]
9 | [F | C]; po; [dsbsy]
10 | [dsb]; po
```

We rewrite the original barrier-ordered-before relation to use the barrier helpers defined in Figure 9.2. The first seven clauses of `bob` capture the same ordering as in the user model, but capturing the barrier hierarchy: imposing the same barrier orderings when using stronger barriers (namely, DSBs in place of DMBs). We use helper sets `F` for all fences (barriers), and `C` for cache operations (including TLBIs).

The Arm DSB instruction does have additional ordering over a DMB, and these are captured by the two new clauses of `bob` above. First, a DSB `SY` orders TLBI instructions and other barriers and cache operations (§8.6.2). Second, all program-order later events must wait for an earlier DSB to finish before performing their explicit memory events.

9.4.4 Translation-ordered-before

```
1 let tob =
2   [T_f]; tfr
3 | [T]; iio; [R|W]; po; [W]
4 | speculative; trfi
```

Translation table walks themselves impose ordering on the surrounding events, in three possible ways:

- ▷ coherence of translation-reads of invalid entries;
- ▷ might-be-same-address for program-order-later accesses; and
- ▷ non-forwarding of the speculative writes.

Invalid writes Reads of invalid entries must not have come from the TLB (§8.3.3). Therefore, for a translation fault, the translation-read of the invalid entry must have come from the coherence-latest write from memory at the time the translation happened. This is what the first clause of `tob` captures: that any translation-reads which read an invalid entry must happen before any writes coherence after the one it read from.

There is a careful subtlety here: we cannot simply include all of `tfr` after a translation-read of invalid, as a thread-local write may be forwarded to the translation table walker before it has propagated to memory (§8.4.4).

Speculation As we saw earlier, speculation interacts with translation in two ways: first, it is forbidden to read-from a still speculative write (§8.4.5), and, second, events program-order-after an instruction which does a translation table walk are speculative until the translation table walk completes (§8.4.1).

To capture these we first define when one event is considered speculative until another event happens, with a new relation, `speculative`:

```
1 let speculative = ctrl | addr; po | [T]; instruction-order
```

4141 This captures all the control-flow dependencies that we model here, the classic `ctrl` and `addr; po`, as well
4142 as a new general `[T]; instruction-order` which says that all events ordered `(iio|po)+` after a translation
4143 read are speculative until the translation read satisfies.

4144 We then include `speculative` to same-thread translation-reads-from (`trfi`) in `ob`, forbidding forwarding
4145 of still-speculative writes to translation table walks. For now, we are unable to give a precise bound on
4146 the ordering for thread-local forwarding, and this area is still currently under investigation with Arm,
4147 including potentially being strengthened to forbid this entirely.

4148 **Might-be-same-address** Finally, we include `[T]; iio; [M]; po; [W]`, which captures that writes cannot
4149 propagate until program-order-earlier instructions have determined their physical addresses (and so will
4150 not fault). Although this edge is subsumed by the `speculative; [W]` edge in `dob`, it is kept here for
4151 clarity.

4152 9.4.5 Contextually-ordered-before

```
1 let ctxob =  
2   speculative; [MSR]  
3   | [CSE]; instruction-order  
4   | [ContextChange]; po; [CSE]  
5   | speculative; [CSE]  
6   | po; [ERET]; instruction-order; [T]
```

4153

4154 The contextually-ordered-before relation, `ctxob`, captures the orderings required from context-changing
4155 and context-synchronising operations, without trying to capture the full extent of the relaxed behaviours.
4156 See [Part III](#) for a more detailed discussion of the ordering from context-synchronisation on an exception.
4157 The `ctxob` relation here is an approximation: capturing the orderings we know the architecture must
4158 provide, but perhaps not exploring the full architectural envelope of possible behaviours to its full extent.

4159 **Speculation** The first guarantee we see is that context changes and synchronisation should not happen
4160 speculatively. Speculative context changes may create translation table roots and associated translation
4161 table walks from unreachable writes, creating thin-air problems (§8.8.1). To prevent this, we ensure that
4162 context-changing operations only happen once they are non-speculative, by enforcing `speculative; [MSR]`
4163 in `ob`. Forbidding the speculative execution of context-synchronising operations is achieved by the inclusion
4164 of `speculative; [CSE]` in `ob`.

4165 **Context synchronising** Context-synchronising events (such as from ISB and ERET instructions) guar-
4166 antee that program-order-earlier context-changing events are seen by program-order-later instructions.
4167 Microarchitecturally, context synchronisation can be achieved by simply flushing the pipeline, restarting
4168 all program-order-later instructions. For now, this effect seems fixed in the architecture (§8.7), and so
4169 we get `[CSE]; instruction-order` in `ob`, subsuming the earlier ISB orderings. To ensure that context
4170 changes are seen after the synchronisation, we include `[ContextChange]; po; [CSE]` in `ob`. The union of
4171 these two relations ensures the context change is ordered before any program-order-later events.

4172 **Exceptions** Taking and returning from exceptions are context synchronising (§8.7). However, translation
4173 reads of a lower exception level should not satisfy during execution at a higher exception level. We over-
4174 approximate this by including `po; [ERET]; instruction-order; [T]` in `ob`, ensuring all translation-reads
4175 after an ERET wait.

9.4.6 Fault-ordered-before and ETS

Note: ETS is subject to change, as noted in §8.4.3, p.123.

```
1 (* ordered-before a fault *)
2 let obfault =
3   data; [FaultFromW]
4 | speculative; [FaultFromW]
5 | [dmbst]; po; [FaultFromW]
6 | [dmbld]; po; [FaultFromW|FaultFromR]
7 | [A|Q]; po; [FaultFromW|FaultFromR]
8 | [R|W]; po; [FaultFromReleaseW]
9
10 (* ETS-ordered-before *)
11 let obETS =
12   (obfault; [Fault_T]); iio-1; [T_f]
13 | ([TLBI]; po; [dsb]; instruction-order; [T]) & tlb-affects
```

To capture the specific guarantees described by FEAT_ETS (§8.4.3, §8.6.2), we include `Fault` events in the candidate executions. These events sit in the execution (in `po` order) where the explicit memory event would have been if there was no fault, tagged with the kind of fault it was (translation or permission).

Ordering to a fault To fully capture the strength of FEAT_ETS, we keep track of syntactic dependencies *into* the instruction which faulted, and apply those dependencies to the `Fault` event itself.

`obfault` is then the syntactic subset of local ordering in `ob`, where the right-hand side of each clause is substituted with a `Fault_T` (a translation fault). We use this to construct an `obETS` relation, whose first clause adds to `ob` exactly this ordering to the translation read of the invalid entry, as architected by FEAT_ETS.

Note that dependencies and orderings *from* a faulting instruction are not required to be respected, and so we do not induce orderings from a `Fault_T`.

FEAT_ETS and TLBI The second clause of `obETS` captures a second architected behaviour of FEAT_ETS: faults after thread-local TLBIs do not need context synchronisation to be ordered after the TLBI. Note that one still needs a DSB to complete the TLBI in that case.

9.4.7 TLBI-ordered-before

```
1 (* ordered-before TLBI *)
2 let obtlbi =
3   obtlbi_translate
4 | [R|W|Fault_T]; iio-1; (obtlbi_translate & ext); [TLBI]
```

Finally, there is the `obtlbi` relation, which captures the ordering between translations, their explicit memory events, and the TLB invalidations which affect them. The relation is split in two: the first clause enforces order between stale translations and the TLBIs they are invalidated by; the second clause imposes additional ordering on the intra-instruction-later explicit events, caused by broadcast TLBIs (§8.6.3).

4199 **Identifying stale TLB entries**

```

1 let tlb_barriered =
2   ([T]; tfr; wco; [TLBI]) & tlb-affects-1

```

4200

4201 When a translation read happens, it is allowed for it to read from a stale write (§8.5.1). That is,
 4202 the translation need not be ordered before writes which come after the write it actually reads from.
 4203 Consequently the `tfr` relation is not included in `ob`.

4204 To account for TLB maintenance, we include some edges from translations to TLBIs, when there is an
 4205 interposing newer write. The general shape of this ordering, named `tlb_barriered` in the model, is
 4206 illustrated in Figure 9.5.

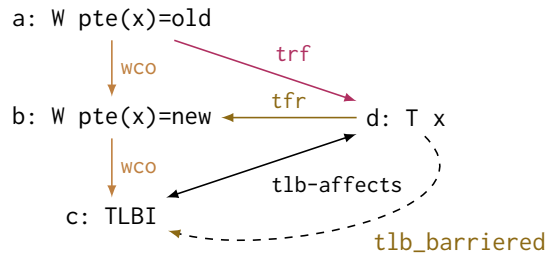


Figure 9.5: General `tlb_barriered` shape.

4207 The `tlb_barriered` auxiliary relation relates a translation read (`d`) to any TLBI (`c`) which targets that
 4208 translations context (ASID, VMID, address, etc) and which is `wco`-after an interposing write (`b`) since the
 4209 write the translation read read from. Intuitively, ‘after’ the TLBI the stale writes will no longer be in the
 4210 TLB, and so translation-reads should not read from them any more.

4211 **Stale translation reads** We can then use this `tlb_barriered` relation to define ordering between
 4212 translation reads and the TLB maintenance which affect them. For stage 2 translations, `tlb_barriered`
 4213 alone would be too strong: since invalidations of second-stage entries also require stage 1 invalidations to
 4214 clear any end-to-end virtual-to-physical mappings (§8.6.4) we must include some additional restrictions,
 4215 considering the stage 1 and stage 2 cases separately.

```

1 (* translate ordered-before TLBI *)
2 let obtlbi_translate =
3   [T & Stage1]; tlb_barriered; [TLBI-S1]
4
5 | (([T & Stage2]; tlb_barriered; [TLBI-S2])
6   ; wco?; [TLBI-S1]
7   )
8   & (same-translation; [T & Stage1]; maybe_TLB_cached)
9
10 | ([T & Stage2]; tlb_barriered; [TLBI-S2])
11   & (same-translation; [T & Stage1]; trf-1; wco-1)

```

4216

4217 For stage 1 translation reads we can include `tlb_barriered` between stage 1 translation-reads and TLBIs
 4218 directly.

4219 For stage 2 translation reads, we have to case split on the execution: either (1) the translation table
 4220 walk does a stage 1 translation read which reads-from an older write, in which case there may have been
 4221 a whole cached translation that must be invalidated; or (2) one of the stage 1 translation reads of the
 4222 translation table walk reads from a write that is newer than the stage 2 TLBI, and so there cannot have
 4223 been any cached whole translation entries in the TLB, and so we only need the stage 2 invalidation. These
 4224 cases are illustrated in Figure 9.6, and correspond to the two clauses of `obtlbi_translate` which match
 4225 on stage 2 translation reads.

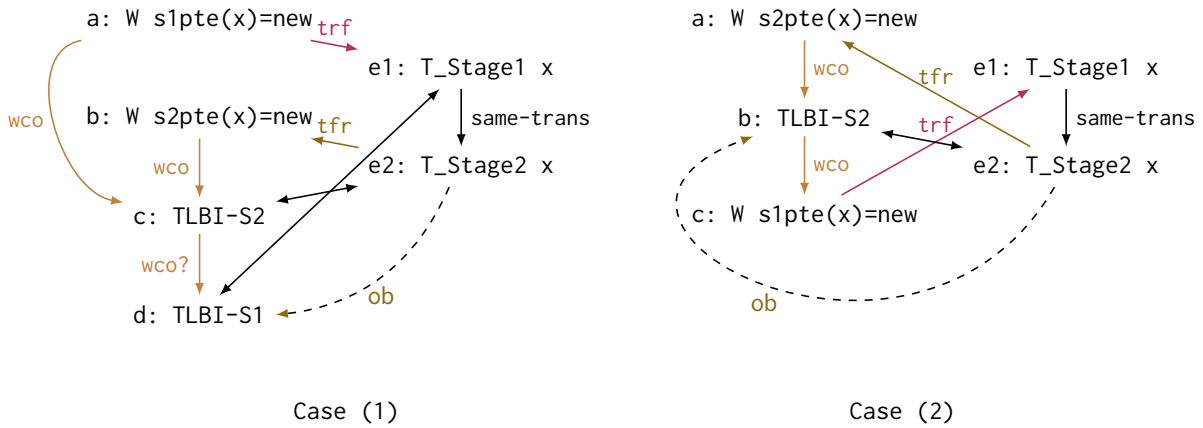


Figure 9.6: obtlbi stage 2 scenarios.

4226 The staggered two-step invalidation in case (1), where a translation-read may have been cached in the
 4227 TLB, is captured with the following `maybe_TLB_cached` relation:

```

1 let maybe_TLB_cached =
2   ([T]; trf-1; wco; [TLBI]) & tlb-affects-1

```

4228

4229 We then use this relation to add ordering from a stage 2 translation-read to the stage 1 TLBI, `wco`-after a
 4230 stage 2 TLBI that removed any stale IPA mappings, which would remove any cached whole-translation
 4231 any stage 1 translation-read might have read from, and after which any fresh translation table walk would
 4232 be required to not see the stale stage 2 entry the translation-read read from.

4233 We capture the general shape of (2) by ordering the second-stage translation-read with the second-stage
 4234 TLBI using `tlb_barriered` just as we did for Stage 1, but only when one of the same-translation stage 1
 4235 walk translation-reads already read from something newer — and therefore there cannot have been a
 4236 whole-translation cached in the TLB.

4237 **Broadcast TLBIs** Recall that broadcast TLBIs impose restrictions on other threads (§8.6.3). When a
 4238 broadcast TLBI's invalidation affects a translation on another core, then it must also affect the explicit
 4239 memory effect associated with it. This shape is illustrated in Figure 9.7, and corresponds to the final
 4240 clause of `obtlbi`.

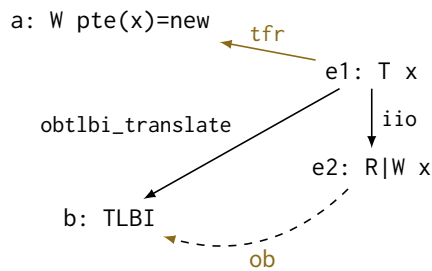


Figure 9.7: obtlbi broadcast TLBI shape.

4241 **Connecting TLB invalidations to translation reads** The final part of the puzzle is how to relate TLBI
4242 events with translations which may be affected by the invalidation. Recall that the TLBIs are grouped
4243 into subsets of TLBI-S1, TLBI-VA, and so on. We define a `tlb_might_affect` that is the cross-product of
4244 these with the `same-*` relations:

```
1 let tlb_might_affect =
2   [ TLBI-S1 & ~TLBI-S2 & TLBI-VA & TLBI-ASID & TLBI-VMID]
3   ; (same-va-page & same-aside & same-vmid) ; [T & Stage1]
4 | [ TLBI-S1 & ~TLBI-S2 & ~TLBI-VA & TLBI-ASID & TLBI-VMID]
5   ; (same-aside & same-vmid) ; [T & Stage1]
6 | [ TLBI-S1 & ~TLBI-S2 & ~TLBI-VA & ~TLBI-ASID & TLBI-VMID]
7   ; same-vmid ; [T & Stage1]
8 | [~TLBI-S1 & TLBI-S2 & TLBI-IPA & ~TLBI-ASID & TLBI-VMID]
9   ; (same-ipa-page & same-vmid) ; [T & Stage2]
10 | [~TLBI-S1 & TLBI-S2 & ~TLBI-IPA & ~TLBI-ASID & TLBI-VMID]
11   ; same-vmid ; [T & Stage2]
12 | [ TLBI-S1 & TLBI-S2 & ~TLBI-IPA & ~TLBI-ASID & TLBI-VMID]
13   ; same-vmid ; [T]
14 | ( TLBI-S1 & ~TLBI-IPA & ~TLBI-ASID & ~TLBI-VMID)
15   * (T & Stage1)
16 | ( TLBI-S2 & ~TLBI-IPA & ~TLBI-ASID & ~TLBI-VMID)
17   * (T & Stage2)
```

4245

4246 Finally, we get `tlb-affects` by attaching `tlb_might_affect` to events in the same thread, and if a TLBI-IS,
4247 to events in other threads too:

```
1 let tlb-affects =
2   ([~TLBI-IS]; tlb_might_affect) & int
3 | [TLBI-IS]; tlb_might_affect
```

4248

```

1 declare wco(Event, Event): bool
2
3 (* wco has domain and range of W,CacheOp *)
4 assert forall ev1: Event, ev2: Event =>
5     wco(ev1, ev2) -->
6     (W(ev1) | C(ev1) | (ev1 == IW)) & (W(ev2) | C(ev2))
7
8 (* wco is transitive *)
9 assert forall ev1: Event, ev2: Event, ev3: Event =>
10    wco(ev1, ev2) & wco(ev2, ev3) --> wco(ev1, ev3)
11
12 (* wco is total *)
13 assert forall ev1: Event, ev2: Event, ev3: Event =>
14    wco(ev1, ev3) & wco(ev2, ev3) & ~(ev1 == ev2) -->
15    wco(ev1, ev2) | wco(ev2, ev1)
16
17 (* wco is irreflexive *)
18 assert forall ev1: Event, ev2: Event, ev3: Event =>
19    wco(ev1, ev2) --> ~(ev1 == ev2)
20
21 (* wco is antisymmetric *)
22 assert forall ev1: Event, ev2: Event =>
23    wco(ev1, ev2) --> ~wco(ev2, ev1)
24
25 (* all write/cache-op pairs are wco related *)
26 assert forall ev1: Event, ev2: Event =>
27    W(ev1) & C(ev2) -->
28    wco(ev1, ev2) | wco(ev2, ev1)
29
30 (* wco is consistent with co *)
31 assert forall ev1: Event, ev2: Event =>
32    co(ev1, ev2) --> wco(ev1, ev2)
33
34 (* all C are wco after IW
35  * n.b. all W are wco after IW, because all W are co after IW
36  * and co => wco
37  *)
37 assert forall ev: Event =>
38    C(ev) --> wco(IW, ev)

```

Figure 9.8: wco.cat: isla-cat definition of wco.

Validating the RVM model

10.1 Validation against the architecture

To ensure that the proposed virtual memory model presented in [Chapter 9](#) correctly captures the architectural intent (where known), we engaged in detailed discussions with Arm.

Our model is produced through an iterative process: where the production of interesting litmus tests are guided by hardware testing and surveying of software requirements; the resulting tests are presented to, and discussed with, Arm architects; new and updated models are created using any architectural intent learned from those discussions; and, finally, those new models are validated against hardware and software requirements, informing the production of further litmus tests.

Ideally, we would run this process until a fixed point is reached. However, this is not always practical. We know the model presented in [Chapter 9](#) is incomplete, and more work is needed to further update the models with the extensions to the architecture and further clarified intent.

10.1.1 Clarity of architecture

We claim that the litmus tests presented in [Chapter 8](#) have known architectural intent, and (as will be discussed in the following sections) the presented model correctly captures that intent for those tests.

For some of these behaviours, it seems improbable that the architectural intent would change. Specifically, the guarantees given by the break, break-before-make, and general TLB-maintenance shapes, are fundamental to the security and correctness of modern software, and so are highly unlikely to be weakened over time.

Some of the behaviours arise as consequences of other parts of the design, specifically around TLB fills (§8.5.2), where the strength of the fill itself arises from a historical design of the processors, and not a fundamental software requirement. As modern hardware has advanced, Arm have added features to specifically weaken those areas (such as with FEAT_nTLBPA).

Conversely, many of the relaxed behaviours may see changes as the architecture evolves. We already saw how the introduction of FEAT_ETS strengthened some aspects of the architecture, and features such as ETS are still in-flux, and there seems no reason to believe that Arm have settled on the final design. Hopefully, the questions raised in this work have helped guide Arm in that design, and resulted in a more stable architecture.

10.1.2 Remaining questions and updates

There are a number of places where the model as presented lacks the underlying architectural clarity to yet give more precise bounds on the architectural envelope.

There are a few places this is apparent in the model presented here:

- ▷ CONSTRAINEDUNPREDICTABLE behaviours due to TLB conflicts (break-before-make violations).
- ▷ Architectural features such as FEAT_nTLBPA, FEAT_ETS2, FEAT_TTL, and FEAT_BBM.

- 4284 ▷ Caching of access permissions, memory types, shareability, and so on.
- 4285 ▷ Sharing TLBs between PEs.
- 4286 ▷ Caching of non-last-level block entries in the TLB.

4287 The first, the constraints on unpredictability, were already discussed earlier (§8.6.5), and more discussion
4288 with architects is required to be able to present a model with any confidence.

4289 The last one (caching of non-last-level block entries) is more interesting, and represents a gap in the
4290 model presented in the previous chapter. When a block entry is cached in the TLB, the hardware has a
4291 choice between caching entries per-page or only one for the whole block. The model currently is too weak,
4292 allowing separately cached entries per-page, and the architectural intent is now clearly to ensure that
4293 TLB invalidations would remove any cached entries for the whole block.

4294 10.2 Validating against hardware

4295 Hardware testing is an important aspect in gaining confidence in any relaxed memory model: without
4296 thorough evaluation of a range of microarchitecture it would not be possible to make strong claims of
4297 soundness of such a model. However, testing systems-level features on hardware is much more challenging
4298 than testing the features covered in previous user-level models (including instruction fetch, as the required
4299 cache maintenance instructions were all unprivileged). Testing virtual memory requires a setup running
4300 at least at EL1, both to be able to run the TLB maintenance instructions, and to enable catching of any
4301 generated exceptions.

4302 One approach would be to use `klitmus7`, an experimental version of `litmus` which produces a kernel
4303 module that runs at EL1 [108]. `klitmus` was primarily designed for the testing of the Linux kernel memory
4304 model, with the kernel modules it produces run as part of the Linux kernel. Attempting to modify the
4305 currently in-use translation tables or exception vectors would interfere with Linux's operations. Using
4306 `klitmus` would therefore require a custom kernel as well as test infrastructure.

4307 Instead, we build a brand new test harness designed for running tests which use systems features such as
4308 TLB maintenance and exception handlers: `system-litmus-harness`¹.

4309 **Limitations** `system-litmus-harness` has some limitations, for now: (1) the harness runs at EL1 and
4310 cannot run tests at EL2; (2) we do not check for known CPU errata for the device being ran on, instead
4311 relying on defensive programming; (3) while the harness can run with QEMU/KVM on any device,
4312 running it bare metal (without a VM) is supported on only a limited number of devices; and (4) the
4313 harness currently uses an ad-hoc litmus test format which is not unified with either `isla-axiomatic` or
4314 `litmus7` itself. We do not believe any of these limitations are fundamental; they should all be solvable
4315 with additional engineering resources.

4316 10.2.1 Harness overview

4317 At its core, `system-litmus-harness` is a relatively simple micro-kernel running at EL1. It builds in a set
4318 of litmus tests, with fixed code for each thread, and an initial state described in an ad-hoc language. The
4319 user gives the harness arguments, at boot, containing the name(s) of litmus tests to run and other run
4320 configuration options. The harness then runs the litmus tests, collects the results, and echos those results
4321 back to the user through the serial output.

4322 The structure of the test runner inside the harness is in a typical *litmus* style. It runs the tests in batches,
4323 executing each thread in a loop, where each iteration of the loop operates on a different set of locations,
4324 making each iteration independent from one another. This is extended in the obvious way for translation,
4325 making each iteration use its own translation tables and ASID.

4326 **Litmus test format** Figure 10.1 gives an example litmus test, `CoTR.inv+dsb-isb`, a variation on the
4327 straight-forward CoRR coherence shape but for translation walks, in the `system-litmus-harness` format.
4328 Litmus tests are dedicated C files which define a `litmus_test_t` struct containing the litmus test data.
4329 The test displayed here can be found at [https://github.com/rem-s-project/system-litmus-harness/
4330 blob/master/litmus/litmus_tests/pgtable/CoTR.inv%2Bdsb-isb.c](https://github.com/rem-s-project/system-litmus-harness/blob/master/litmus/litmus_tests/pgtable/CoTR.inv%2Bdsb-isb.c).

¹<https://github.com/rem-s-project/system-litmus-harness>

```

1 #include "lib.h"
2
3 #define VARS x, y
4 #define REGS p1x0, p1x2
5
6 static void P0(litmus_test_run* data)
7 {
8     asm volatile (
9         /* setup */
10        "mov x0, %[ydesc]\n\t"
11        "mov x1, %[xpte]\n\t"
12        /* code */
13        "str x0, [x1]\n\t"
14        :
15        : ASM_VARS(data, VARS),
16        ASM_REGS(data, REGS)
17        : "cc", "memory", "x0", "x1"
18        );
19 }
20
21 static void sync_handler(void)
22 {
23     asm volatile (
24        "mov x0, #0\n\t"
25
26        ERET_TO_NEXT(x10)
27    );
28 }
29
30 static void P1(litmus_test_run* data)
31 {
32     asm volatile (
33        /* setup */
34        "mov x1, %[x]\n\t"
35        "mov x3, %[xpte]\n\t"
36        /* code */
37        "ldr x0, [x1]\n\t"
38        "dsb sy\n\t"
39        "isb\n\t"
40
41        "ldr x2, [x3]\n\t"
42        /* teardown */
43        "str x0, [%[outp1r0]]\n\t"
44        "cbz x2, .after\n\t"
45        "mov x2, #1\n\t"
46        ".after:\n\t"
47        "str x2, [%[outp1r2]]\n\t"
48        :
49        : ASM_VARS(data, VARS),
50        ASM_REGS(data, REGS)
51        : "cc", "memory", "x0", "x1",
52        "x2", "x3", "x10"
53    );
54 }
55 litmus_test_t CoTRinv_dsbisb = {
56     "CoTR.inv+dsb-isb",
57     MAKE_THREADS(2),
58     MAKE_VARS(VARS),
59     MAKE_REGS(REGS),
60     INIT_STATE(
61         2,
62         INIT_UNMAPPED(x),
63         INIT_VAR(y, 1)
64     ),
65     .interesting_result = (u64[]){
66         /* p0:x0 ==*/1,
67         /* p0:x2 ==*/0,
68     },
69     .thread_sync_handlers =
70     (u32**[]){
71         (u32*[]){NULL, NULL},
72         (u32*[]){(u32*)sync_handler,
73             NULL},
74     },
75     .requires_pgtable = 1,
76     .no_sc_results = 3,
77 };

```

Figure 10.1: CoTR.inv+dsb-isb litmus test, system-litmus-harness source.

4331 The header VARS and REGS define the global variables to allocate (in this case, we want two, named x and
4332 y), and the names of output variables (which we usually style after the names of the machine registers
4333 which store them) for the final register values to save from the test.

4334 The test then defines two threads with two static functions, P0 and P1, containing the code of the threads.
4335 These functions take a litmus_test_run struct, which contains the virtual addresses of each of the global
4336 and output variables, and any other initial state required for the test.

4337 Taking the code for Thread 1, in P1, as an example, it is given as an asm block which contains the test
4338 code sandwiched between some setup and teardown code that moves values from the C code into the
4339 machine registers the test uses, and back out at the end.

4340 This test has an exception handler for this thread. It is given by the sync_handler function and set as
4341 the vector for this thread in the initial state. The handler simply resets x0 to 0, and then performs an
4342 ERET to the next instruction address (that is, to ELR+4).

4343 The final block of the test is the litmus_test_t struct, which gives the C definition for the test. It
4344 provides the name, the number of threads, the global and output variables, which exception handlers to
4345 install for each thread, the particular relaxed result to mark, and the initial machine state to run the test
4346 from. In this case, the initial state says that x starts unmapped (invalid at level 3), and y is mapped to a
4347 location that contains the value 1. Implicitly, global variables have virtual addresses in distinct pages.

4348 Litmus test format reference

4349 Our test format supports writing a variety of kinds of pagetable tests with different initial state setups.
4350 Appendix C describes the test format in full.

4351 As an example, take the INIT_STATE from the ROT1+dsb-dsb-tlbi-dsb test¹, which defines three variables:
4352 x, y, and z. Its initial state is reproduced in Figure 10.2. It says that all three variables start out mapped
4353 with initial values 0, 1, and 2, respectively (L13-15). Next, it tells the allocator that x should be allocated
4354 in its own 2MiB region (L16), but to nevertheless place y in that region too (L17) with the same page
4355 offset, i.e. it should have the same least significant 12 bits as x (L18). Finally, it tells the allocator to
4356 place z in its own 2MiB region, with the same PMD offset (bits 20-12) as x (L20). This ensures that bits
4357 12-0 overlap for x and y, and bits 20-12 overlap for x and z, and therefore the table containing the entry
4358 for y can be assigned to the level 2 entry for x, as required by the ROT test shape (see §8.4.8).

```
1 #define VARS x, y, z
2 #define REGS p0x4
3
4 /* see source for full test */
5
6 litmus_test_t ROT1_dsbtlbidsb = {
7     "ROT1+dsb-dsb-tlbi-dsb",
8     MAKE_THREADS(1),
9     MAKE_VARS(VARS),
10    MAKE_REGS(REGS),
11    INIT_STATE(
12        8,
13        INIT_VAR(x, 0),
14        INIT_VAR(y, 1),
15        INIT_VAR(z, 2),
16        INIT_REGION_OWN(x, REGION_OWN_PMD),
17        INIT_REGION_PIN(y, x, REGION_SAME_PMD),
18        INIT_REGION_OFFSET(y, x, REGION_SAME_PAGE_OFFSET),
19        INIT_REGION_OWN(z, REGION_OWN_PMD),
20        INIT_REGION_OFFSET(z, x, REGION_SAME_PMD_OFFSET),
21    ),
22    .interesting_result = (u64[]){
23        /* p0:x2 ==*/1,
24    },
25    .start_els = (int[]){1},
26    .requires_pgtable = 1,
27    .no_sc_results = 2,
28 };
```

Figure 10.2: system-litmus-harness initial state for an ROT-shaped test.

4359 10.2.2 Results from hardware

4360 We ran a collection of hand-written litmus tests on three hardware devices using system-litmus-harness
4361 running inside KVM: a Raspberry Pi 3B+ (Arm Cortex-A53 r0p4), Raspberry Pi 4B (Arm Cortex-A72
4362 r0p3), and an AWS m6g-metal instance (claiming to be an A76).

4363 Note that the hardware tests represent an overlapping set of tests with those presented in Ch.8: some
4364 contain BBM violations; some tests are not reproduced on hardware; and some may appear with
4365 different names, e.g. CoWTf.inv+dmb test (Figure 8.18, p.122) appears in the table as CoWT.inv+dmb.
4366 Tables 10.1 and 10.2 list the total results for all the tests from all three devices.

4367 Our testing revealed some incompatibilities between the architectural intent and the current implementa-
4368 tions. For some break-before-make sequences, such as test MP.BBM1+dsb-tlbiis-dsb-dsb-isb+dsb-isb we

¹which can be found at https://github.com/rem-s-project/system-litmus-harness/blob/master/litmus/litmus_tests/pgtable/pmds/ROT1%2Bdsb-dsb-tlbi-dsb.c

4369 did observe some rare violations of the architectural intent on the Arm cortex-based cores in the Raspberry
4370 Pi 4B and AWS Graviton cores¹. The related MP.BBM1+[dmb.ld]-tlbiis-dsb-isb-dsb-isb+dsb-isb test
4371 (with a detour after the write) was never observed however, suggesting it is related to the DSB not fully
4372 propagating the store, which implies it may be related to other known CPU errata. These anomalous
4373 results have been reported, and are under investigation by Arm.

4374 10.3 Validation by abstraction

4375 We cannot ‘prove’ that the model is correct. Correctness of a relaxed memory model like this depends on
4376 the architects’ intent, and that may change as new revisions of the architecture are released. However, we
4377 can identify properties we believe *any* sound model would have, and check that the model presented here
4378 has those properties.

4379 The key property is that the presented model has a ‘virtual memory abstraction’. While there is no general
4380 definition of what such an abstraction is, we give one intuitive and informal definition: a program with a
4381 fixed injective translation table mapping behaves as if executing above physical memory directly. We can
4382 state this virtual memory abstraction as a property over candidate executions.

4383 To do this, we define a *translation erasure* operation: given a candidate C , the translation-erased candidate
4384 $C^{\sim T}$ is C , but where all TLBI, τ , and τ_f events are erased; any edge containing such events as source or
4385 target removed; and extended with the derived relations `addr` and `po` from C .

4386 If given a full (with all the translation table walk events) well-formed (consistent with the intra-instruction
4387 semantics) candidate C , with no TLBI events, no τ_f events, and no `W` events to any pagetable location,
4388 then, the candidate is consistent in the VMSA model if and only if the translation-erased candidate $C^{\sim T}$
4389 is consistent in the base model.

4390 Informally, the proof is a straightforward inclusion proof by relation algebra. The internal and atomic
4391 axioms are trivially subset inclusions of one another under translation erasure. Additionally, the translation-
4392 internal relation is trivially a subset of the usual internal one with translation events erased. For external,
4393 we show that `ob` in the base model implies `ob` in the VMSA model, and that `ob` in the VMSA model
4394 implies the same `ob` in the base model. Therefore they must forbid the same cycles. See Appendix D for a
4395 complete proof, largely due to Jean Pichon-Pharabod.

¹Further investigation indicates that our `m6g-metal` instance may have in-fact also been a mislabelled Graviton1, also based on the Arm Cortex A72.

Table 10.1: system-litmus-harness hardware results from three devices: Part I.

Name	rpi4b	rpi3bp	graviton2
CoRT	964.72K/8M	520.06K/3M	2.29M/108M
CoRT+dsb-isb	802.86K/8M	327.02K/3M	3.41M/108M
CoTR	2.51M/8M	0/3M	21.70M/107.50M
CoTR+addr	0/8M	1/3M	0/107.50M
CoTR+dmb	1/8M	0/3M	4/107.50M
CoTR+dsb	2/8M	0/2.50M	5/107M
CoTR+dsb-isb	1/8M	0/2.50M	1/107M
CoTR.inv	3.63M/6.50M	0/2.50M	32.28M/43M
CoTR.inv+dsb-isb	0/6.50M	0/2.50M	0/43M
CoTR1+dsb-dc-dsb-tlbi-dsb-isb	2/6.50M	0/2.50M	4/43M
CoTR1+dsb-tlbi-dsb-isb	2/6.50M	0/2.50M	3/43M
CoTR1.tlbi+dsb-isb	6/6.50M	1/2.50M	29/43M
CoTT	0/6.50M	0/2M	0/43M
CoTW	0/1.50M	0/1.50M	0/10.50M
CoWT	3.77M/6.50M	1.85M/2M	22.64M/43M
CoWT+dsb	3.76M/6.50M	995.06K/2M	21.50M/43M
CoWT+dsb-isb	3.78M/6.50M	995.77K/2M	21.50M/43M
CoWT+dsb-svc-tlbi-dsb	0/6.50M	0/2M	0/42.50M
CoWT.inv	10/6.50M	1.73M/2M	169/42.50M
CoWT.inv+dmb	8/6.50M	69.38K/2M	42/42.50M
CoWT.inv+dsb	1/6.50M	0/2M	57/42M
CoWT.inv+dsb-isb	0/6.50M	0/2M	0/42M
CoWT1+dsb-tlbi-dsb	0/6.50M	0/2M	0/42.50M
CoWT1+dsb-tlbi-dsb-isb	0/6.50M	0/2M	0/42.50M
CoWinvT	4.17M/6.50M	1.79M/2M	26.81M/42M
CoWinvT+dsb-isb	4.19M/6.50M	1.83M/2M	26.80M/42M
CoWinvT1+dsb-tlbi-dsb	0/6.50M	0/2M	0/42M
CoWinvT1+dsb-tlbi-dsb-dsb-isb	0/6.50M	0/2M	0/42M
ISA2.TRR+dmb+po+dmb	0/6.50M	0/2M	0/42M
MP.BBM1+[dmb.ld]-dsb-tlbiis-dsb-isb-dsb-isb+dsb-isb	0/108.50M	0/1.50M	0/437.50M
MP.BBM1+[dmb.ld]-tlbiis-dsb-isb-dsb-isb+dsb-isb	0/198.50M	0/1.06G	0/129.50M
MP.BBM1+[po]-dsb-tlbiis-dsb-isb-dsb-isb+dsb-isb	0/108.50M	0/1.50M	0/145.50M
MP.BBM1+dsb-isb-tlbiis-dsb-isb-dsb-isb+dsb-isb	0/6.50M	0/2M	52/135.50M
MP.BBM1+dsb-tlbiis-dsb-dsb+dsb	1/6.50M	0/2M	7/42.50M
MP.BBM1+dsb-tlbiis-dsb-dsb+dsb-isb	0/6.50M	0/2M	2/42.50M
MP.BBM1+dsb-tlbiis-dsb-dsb-isb+dsb	1/6M	0/2M	0/42.50M
MP.BBM1+dsb-tlbiis-dsb-dsb-isb+dsb-isb	2/6M	0/2M	3/42.50M
MP.BBM1+po-dsb-tlbiis-dsb-isb-dsb-isb+dsb-isb	0/1M	0/1.50M	9/191.50M
MP.BBM1.id+dsb-tlbiis-dsb-dsb+dsb-isb	10/6M	2/2M	87/42.50M
MP.RT+svc-dsb-tlbi-dsb+dsb-isb	1/6M	0/2M	3/42M
MP.RT+svc-dsb-tlbiis-dsb+dsb-isb	1/6M	0/2M	3/42M
MP.RT.inv+dmb+addr	0/6M	0/2M	0/42M
MP.RT.inv+dmb+po	0/6M	6/1.50M	0/42M
MP.RT1+[dmb.ld]-dmb+dsb-isb	7.15K/6M	986/1.50M	1.26K/42M
MP.RT1+[dmb.ld]-dsb-isb-tlbiis-dsb-isb+dmb	0/1M	0/1M	0/23M
MP.RT1+[dmb.ld]-dsb-isb-tlbiis-dsb-isb+dsb-isb	0/1M	0/1M	0/23M
MP.RT1+[dmb.ld]-dsb-tlbiis-dsb-isb+dmb	0/6M	0/1.50M	0/42M
MP.RT1+dc-dsb-tlbiis-dsb+dsb-isb	4/6M	1/1.50M	5/41.50M
MP.RT1+dc-dsb-tlbiis-dsb-isb+dsb-isb	3/6M	0/1.50M	2/41.50M
MP.RT1+dsb-isb-tlbiis-dsb-isb+dsb-isb	0/6M	0/1.50M	4/41M
MP.RT1+dsb-tlbi-dsb+dsb-isb	0/6M	0/1.50M	2/41M
MP.RT1+dsb-tlbiis-dsb+dsb-isb	5/6M	0/1.50M	6/41M
MP.RT1+dsb-tlbiis-dsb+dsb-isb	3/6M	0/1.50M	2/41M
MP.RT1+dsb-tlbiis-dsb+dsb-isb	1/6M	0/1.50M	1/41M

Table 10.2: system-litmus-harness hardware results from three devices: Part II.

Name	rpi4b	rpi3bp	graviton2
MP.RT1+dsb-tlbiis-dsb-isb+dmb	0/6M	0/1.50M	1/41M
MP.RT1+dsb-tlbiis-dsb-isb+dsb-isb	0/6M	0/1.50M	1/41M
MP.RT1+dsb-tlbiis-dsb-tlbiis-dsb+dsb-isb	0/6M	0/1.50M	3/41M
MP.TT+Winv-dmb-Winv+tpo	254.83K/6M	114.48K/1.50M	170.96K/41M
MP.TT+dmb+dsb-isb	688.65K/5.50M	174.78K/1.50M	492.98K/41M
MP.TT+dmb+tpo	843.79K/5.50M	157.80K/1.50M	480.31K/41M
MP.TT.inv+dmb+dsb-isb	0/5.50M	0/1.50M	0/41M
MP.TT.inv+dmb+tpo	0/5.50M	0/1.50M	0/41M
MP.invRT+dsb+dsb-isb	871.53K/5M	101.75K/1.50M	1.78M/40.50M
MP.invRT1+dsb-isb-tlbiis-dsb-isb+dsb-isb	0/5.50M	0/1.50M	1/41M
MP.invRT1+dsb-tlbiis-dsb+dsb	0/5M	0/1.50M	2/41M
MP.invRT1+dsb-tlbiis-dsb+dsb-isb	1/4.50M	0/1.50M	1/41M
WRC.AT+ctrl+dsb	128.64K/4.50M	77.36K/1.50M	214.45K/40M
WRC.TRR+addr+dmb	0/4.50M	0/1.50M	0/40M
WRC.TRR.inv+addrs	0/4.50M	0/1.50M	0/40M
WRC.TRT+addr+dmb	35.28K/4.50M	32.50K/1.50M	103.16K/40M
WRC.TRT+dmbs	53.60K/4.50M	36.76K/1.50M	171.51K/40M
WRC.TRT+dsb-isbs	18.80K/4.50M	30.44K/1.50M	104.62K/39.50M
WRC.TRT.inv+addrs	0/4M	0/1.50M	0/38.50M
WRC.TRT.inv+dsb-isbs	0/4M	0/1M	0/38M
WRC.TRT.inv+po+addr	0/4M	0/1M	0/37.50M
WRC.TRT.inv+po+dmb	0/4M	0/1M	0/37M
WRC.TRT1+dsb-tlbiis-dsb+dmb	0/4.50M	0/1M	0/38M
WRC.TRT1+dsb-tlbiis-dsb+dsb-isb	0/4.50M	0/1M	0/38M
CoWR.alias	0/6M	0/1.50M	0/36M
MP+dmb-data+dmb	0/5M	0/1.50M	0/36M
MP.alias+dmbs	0/5M	0/1.50M	0/36M
MP.alias2+dmb-data+dmb	0/5M	0/1.50M	0/36M
MP.alias2+dmbs	0/3M	0/1.50M	0/19.50M
MP.alias2+po-data+dmb	2.23K/5M	3.17K/1.50M	407.36K/36M
MP.alias3+rfi-data+dmb	51/3M	16/1.50M	36.35K/19.50M
SB.alias+dmbs	0/5M	0/1M	0/35.50M
WRC.alias2+addrs	0/4M	0/43M	0/19M
WRC.alias2+dmbs	0/4M	0/43M	0/18.50M
MP.NC+dsb-dc-dsb-dmb+dmb	138.80K/8M	364.97K/26M	54.95K/25.50M
MP.NC+po-dmb+dmb	345.33K/7.50M	642.90K/25.50M	333.55K/25.50M
MP.NC1+dsb-tlbiis-dsb-dc-dsb-dmb+dmb	0/7.50M	0/25.50M	0/25.50M
MP.NC1+dsb-tlbiis-dsb-dmb+dmb	556/7.50M	482/25.50M	6/25.50M
WR.NC+dsb	0/0	0/0	0/0
WR.NC+po	0/0	0/0	0/0
WR.WARA-NC+dsb	0/0	0/0	0/0
WR.WARA-NC+po	0/0	0/0	0/0
WWR.NC+po-po	0/0	0/0	0/0
CoWT.L23+dsb-isb	11.45M/13M	6.73M/13.50M	48.94M/84.50M
CoWT.L23+po	12.88M/13M	13.39M/13.50M	80.61M/84.50M
CoWT1.L23+dsb-tlbi-dsb-isb	0/13M	0/13.50M	0/84.50M
ROT+dsb-dsb	0/13M	0/13.50M	0/84.50M
ROT+po-po	0/13M	0/13.50M	0/84M
ROT1+dsb-dsb-tlbi-dsb	0/13M	0/13.50M	0/84M
ROT1+dsb-dsb-tlbivaa-dsb	0/13M	0/13.50M	0/84M
CoTT+dsb-popage	0/35.50M	0/31M	0/1.12G
CoTT+po-popage	1/47M	0/43.50M	0/1.20G
WR.MAIR1+dsb-isb-dc-dsb	0/0	0/0	0/0
WR.MAIR1+dsb-isb-po	0/0	0/0	0/0
WR.MAIR1+dsb-tlbi-dsb-isb-dc-dsb	0/0	0/0	0/0
WR.MAIR1+dsb-tlbi-dsb-isb-po	0/0	0/0	0/0
WR.MAIR1+po-po	0/0	0/0	0/0

Exceptions and interrupts

This part is based on: in-progress and under-submission work done in collaboration with Alasdair Armstrong, Thomas Bauereiss, Brian Campbell, Ohad Kammar, Jean Pichon-Pharabod, and Peter Sewell.

Relaxed precise exceptions

We now turn to the final part, and discuss hardware support for exceptions and interrupts.

We do so in the way previous work have made now typical: we describe the main phenomena and architectural design space, through the exploration of litmus tests; we use those litmus tests as a catalyst for discussions about the architectural intent with the architects and for discovery of the current implementations by the surveying of hardware; we produce a formal mathematical model that captures that intent; and, finally, we validate that model by making it executable as a test oracle and execute a suite of litmus tests, comparing the results to hardware and previously collected intent from architects.

Chapter contents

11.1 Introduction	183
11.1.1 Exception taxonomy	183
11.1.2 Exception lifecycle	184
11.1.3 Vectors and vector tables	184
11.1.4 Precision	185
11.2 Instruction instances	185
11.2.1 Fetch-decode-execute trees and streams	186
11.3 Relaxed behaviour of precise exceptions	187
11.3.1 Out-of-order execution across exception boundaries	187
11.3.2 Context synchronisation and speculation	188
11.3.3 Privilege level	188
11.3.4 Dependency through system registers	191
11.3.5 Ordering from asynchronous exceptions	191
11.3.6 Exception-specific mechanisms	191
11.3.7 Exceptions and the intra-instruction semantics	192
11.3.8 Disabling context synchronisation	192
11.4 Synchronous external aborts	193
11.4.1 Behaviour resulting from synchronous external aborts	193

11.1 Introduction

Hardware exceptions (and their many variants: interrupts, traps, faults, aborts, etc.) provide support for many exceptional situations that systems software has to manage. This includes explicit privilege transitions via system calls, implicit privilege transitions from trappable instructions, inter-processor software-generated interrupts, external interrupts from timers or devices, recoverable faults like address translation faults, and non-recoverable faults like memory error correction faults.

To manage exceptions, software relies on a key architectural guarantee, *precision*: that exceptions appear to execute between instructions. To confidently write concurrent systems code that handles exceptions, e.g. mapping on demand at page faults, programmers need a well-defined and well-understood semantics. These modern definitions of precision (e.g. in the current Arm-A documentation) are mostly unchanged over the last 60 years, dating back to at least the IBM System/360. These definitions fundamentally assume a sequential programmers model. For example, Hennessy and Patterson state [109]:

An exception is imprecise if the processor state when an exception is raised does not look exactly as if the instructions were executed sequentially in strict program order

However, modern architectures with programmer-observable relaxed behaviour, such as Arm-A, make such a naive definition inadequate, and leaves it unclear exactly what guarantees there are on exception entry and exit. On pipelined out-of-order processors with observable speculative execution, exceptions have subtle interactions with relaxed memory behaviour which had not previously been investigated.

Overview In this part, we begin by clarifying the key concepts needed to discuss exceptions in the relaxed-memory setting (§11.1-11.2), through the exploration the basic relaxed behaviour across exception boundaries (§11.3). We extend this by introducing the potential of external aborts and examining how they effect the programmer-visible behaviour (§11.4).

We develop an axiomatic model for precise exceptions on Arm-A, including tooling for executing it as a test oracle, along with a library of tests (Chapter 12).

Finally, we validate this model (Chapter 13) by extending the harness presented in Part II and collecting data from a range of implementations.

11.1.1 Exception taxonomy

Arm-A defines multiple kinds of exception [81, D1.3.1]:

- ▷ Synchronous exceptions. These originate from an instruction, e.g. supervisor/hypervisor calls, traps, data/instruction, page faults, etc.
- ▷ Asynchronous exceptions. These are interrupt requests from other processors/peripherals/timers, or system errors.

In Arm nomenclature, any non-synchronous exception is called an interrupt.

Synchronous exceptions are further broken down into *classes*, for example:

- ▷ PC Alignment, for a misaligned program counter.
- ▷ Instruction abort, for MMU faults on instruction accesses.
- ▷ Undefined instruction encoding.
- ▷ Data abort, for MMU faults on data accesses.
- ▷ Execution of an SVC (supervisor call).
- ▷ Trapped register access, from attempting accessing a register that is not permitted or is configured to trap.

For a complete list of exception classes, and their prioritisation, we refer to the Arm architecture reference manual [81, D1.3.5, p5369].

11.1.2 Exception lifecycle

When an exception is taken, execution jumps to the appropriate *exception vector*. Vectors are pre-determined locations which contain code to be executed on the event of an exception. Different kinds of exception jump to different vectors, and so the currently in-use vectors form a *vector table*. Software configures the vectors by setting the base address of the vector table, by writing to the appropriate vector base address register (VBAR).

On taking the exception:

- ▷ The current processor state is saved into the saved program status register (SPSR). This includes the current exception level, status flags and condition bits, and interrupt masking (described in more detail later).
- ▷ The privilege level typically escalates (e.g. from EL0 to EL1).
- ▷ The program-counter to return to (the ‘preferred return address’) is saved into the appropriate exception link register (ELR).
- ▷ The cause of the exception is saved into either the exception syndrome register (ESR) for synchronous exceptions, telling the programmer the *class* of the exception and other associated data; or into the interrupt status register (ISR), telling the programmer which interrupt(s) are pending.
- ▷ If the exception is a translation-related fault, the faulting address is also saved into the fault address register (FAR).
- ▷ The PC is set to the current VBAR plus the appropriate offset.

The code then executed is termed the *exception handler*. Execution continues in the new state until the processor executes an ERET (‘exception return’) instruction. On executing an ERET:

- ▷ The saved processor state (SPSR) is restored.
- ▷ The value saved in the ELR is written to the PC.

Thus, execution jumps back to where the program was executing before the exception was taken, in much the same processor state as it was in at the time.

Preferred return address

The ‘preferred return address’ of synchronous exceptions has an architecturally defined relationship with the instruction that caused the exception. For most instructions, the preferred return address is the program counter value at the point when the exception is taken, therefore returning back to the same instruction once the exception is handled.

There is one exception to this: the class of *exception generating instructions*, whose sole purpose is to generate a particular kind of exception. The most common of these is the SVC (‘supervisor call’) instruction, which is used to implement system calls. These instructions preferred return address is always the *next* instruction, that is, $PC + 4$.

11.1.3 Vectors and vector tables

The appropriate vector is determined from: the *type* of the exception, either synchronous, interrupt request (IRQ), ‘fast’ interrupt request, or external abort (which is described in more detail later); the current stack pointer in use; whether the exception originates from a lower exception level; and whether the exception originates from the 32-bit mode or not. As such the vector table contains 16 vectors. Each vector is 128 bytes. The vectors are then located at a given offset from the base address, see Figure 11.1.

Exception from	Exception type			
	Synchronous	IRQ	Fast IRQ	External abort
Current EL, using stack pointer SP_EL0	0x000	0x080	0x100	0x180
Current EL, using this EL’s stack pointer	0x200	0x280	0x300	0x380
Lower EL, in 64-bit mode	0x400	0x480	0x500	0x580
Lower EL, in 32-bit mode	0x600	0x680	0x700	0x780

Figure 11.1: Arm vector table offsets [81, D1.3.1].

4513 There is not a single active vector table, but one per exception level, with all the exception-related registers
4514 (SPSR, ELR, ESR, FAR, etc) appropriately banked (with one per exception level).

4515 Note that in Armv8, fast interrupt requests function identically to normal interrupt requests. However,
4516 they have independent routing machinery. Interrupt controllers may freely choose to route different
4517 interrupts as different types, but which type the interrupt is has no effect on the execution of the machine.

4518 11.1.4 Precision

4519 Historically, the introduction of pipelined machines raised issues about the behaviour of exceptions:
4520 since instructions may have already been partially executed, the resulting interrupts would appear as
4521 a discontinuity in the flow of instructions [110]. Since then, hardware has had two kinds of exceptions:
4522 imprecise exceptions retain that discontinuity, whereas precise ones take the performance penalty of
4523 recovering (e.g. by discarding later instructions and restarting earlier instructions) to guarantee more
4524 predictable behaviours that programmers could rely on. Intuitively, for a precise exception one can
4525 pinpoint a particular point in the sequence of instructions where the exception happens.

4526 Today, Arm retains imprecise exceptions, but only in some cases: all synchronous exceptions and interrupt
4527 requests are precise. Only system errors — errors from the external system reported back the CPU
4528 asynchronously — may be imprecise. We discuss external aborts in more detail in §11.4.

4529 11.2 Instruction instances

4530 One often thinks of processors as executing *instructions* in some *instruction sequence*, and common
4531 terminology is based on those two concepts. For example, the Arm manual has around 60 instances of
4532 *instruction stream* or *execution stream*.

4533 However, exceptions can arise at multiple points within the fetch-decode-execute cycle, including during
4534 the fetch and decode, when there is no ‘instruction’. For Armv9.4-A, much of this is captured in an Arm
4535 top-level function written in the Arm Architecture Specification Language (ASL), which previous work by
4536 Armstrong, Campbell et al. [45, 111, 112] translated into Sail. This gives us an executable semantics of
4537 the sequential ISA aspects of Armv9.4-A with exceptions (§13.2).

4538 A highly simplified outline of a single-instruction slice of the (400k line) instruction semantics is given in
4539 Figure 11.2.

```
function __TopLevel() =
  // in TakePendingInterrupts:
  if IRQ then AArch64_TakePhysicalIRQException()
  if SE then AArch64_TakePhysicalSErrorException(...)
  // in AArch64_CheckPCAlignment:
  if pc[1..0] != 0b00 then AArch64_PCAlignmentFault()
  // in __FetchInstr:
  opcode = AArch64_MemSingle_read(pc, 4) // read memory
  // in __DecodeA64:
  match opcode
  [1,_,1,1,1,0,0,1,0,1,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_] =
    // the semantics for one family of instructions,
    // including loads LDR Xt,[Xn]
    // execute_aarch64_instrs_memory_single_general_
    // immediate_signed_post_idx(n,t,...)
    let address = X_read(n, 64) // read register n
    let data : bits('datasize') = // read memory
      Mem_read(address, DIV(datasize,8))
    // write register t
    X_set(t, regsize) = ZeroExtend(data, regsize)
```

Figure 11.2: Outline of a single-instruction slice of the Arm intra-instruction semantics.

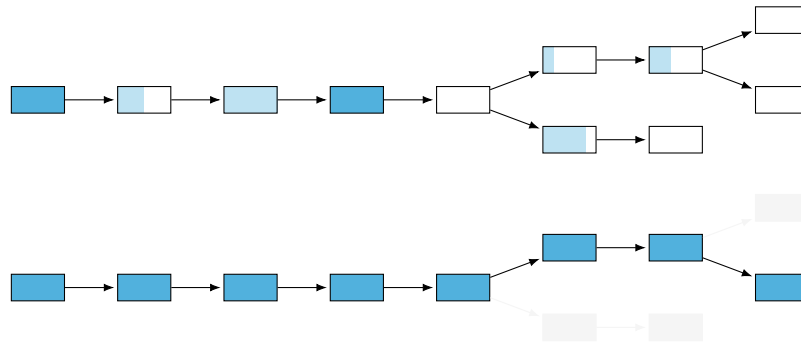


Figure 11.3: Top. The tree of (partially) executed FDX instances at one time, in hardware or operational model execution. **Bottom.** The sequence of architecturally executed FDX instances in a completed execution.

4540 Executing this semantics may lead to one or more kinds of exception, calling the ASL/Sail function
 4541 `AArch64_TakeException()`. This function writes the appropriate values to registers, e.g. computing the
 4542 next PC, exception level, etc. and terminates this `__TopLevel()` execution. So instead of ‘instruction
 4543 instances’, we refer to *fetch-decode-execute instances* (FDX instances), a single execution of `__TopLevel()`.

4544 11.2.1 Fetch-decode-execute trees and streams

4545 One must relate the out-of-order speculative execution of hardware implementations and the architectural
 4546 definition of the allowed behaviours.

4547 At any instant, each core may be processing, out-of-order and speculatively, many instructions (really,
 4548 FDX instances). Partially executed instances are restarted or discarded if they would violate the intended
 4549 semantics (e.g. on a mispredicted branch).

4550 One can visualise the state of a single core abstractly as a tree of partially and completely executed
 4551 instances, as in Figure 11.3 (top). Abstract-microarchitectural operational semantics have long made use
 4552 of this abstraction to implement the thread subsystems [8, 20, 15, 48, 16, 7], see Chapter 2. We now lift
 4553 this model-specific concept into the domain of architecture.

4554 In the figure, we depict the retired (committed) FDX instances as solid dark green, and partially/tentatively
 4555 executed in-flight instances as light green. The arrows depict program order. Committed instances can
 4556 be program-order-after in-flight instances, and non-committed instances may need to be restarted.
 4557 Eventually all FDX instances for this hardware thread will be either committed or discarded, e.g. as
 4558 in Figure 11.3 (bottom). These are the *architecturally executed* FDX instances. The architecture
 4559 definition, and any formal semantics thereof, have to define which such sequences are allowed for each
 4560 thread. This definition includes the register content; memory read values; and their relationships
 4561 with other threads, as determined by the relaxed concurrency model. Axiomatic concurrency models,
 4562 e.g. [13, 113, 114, 115, 116, 117, 4, 2, 34, 118, 69, 42, 32, 46], have candidate executions which contain
 4563 events just from these architecturally executed instances.

4564 The Arm prose specification, given in Figure 11.4 (top), previously attempted to capture the relationship
 4565 between implementation execution (out-of-order and speculative) and the architectural definition of
 4566 allowed behaviour in terms of a notion of a ‘simple sequential execution’ of the machine. As the prose says,
 4567 simple sequential execution does not hold for the intended relaxed-memory architecture. We propose a
 4568 more correct rephrasing that allows for exceptions and other systems phenomena in Figure 11.4 (bottom).

4569 Figure 11.5 depicts a tree of instances involving exception entry (SVC) and return (ERET). Arm-A allows
 4570 implementations to execute the exception handler’s instruction instances out-of-order with respect to
 4571 instances program-order-before the exception entry and program-order-after the exception return. The
 4572 constraints on this freedom is what we now explore.

Architecturally executed An instruction is architecturally executed only if it would be executed in a simple sequential execution of the program. [...]

Simple sequential execution The behavior of an implementation that fetches, decodes and completely executes each instruction before proceeding to the next instruction. Such an implementation performs no speculative accesses to memory, including to instruction memory. The implementation does not pipeline any phase of execution. In practice, this is the theoretical execution model that the architecture is based on, and Arm does not expect this model to correspond to a realistic implementation of the architecture.

Architecturally executed A candidate execution can be architecturally executed if it is composed of a sequence of FDX instances for each thread that together satisfy the Arm concurrency model [extended to cover exceptions, as described here, and other systems phenomena], starting from the machine initial state.

Figure 11.4: Arm prose specification [81, Glossary, p12916] (**top**) and our suggested rephrasing (**bottom**).

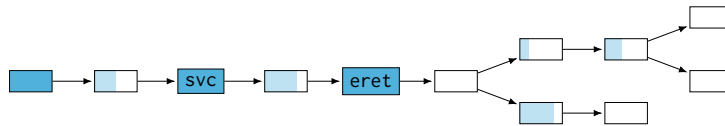


Figure 11.5: The tree of partially and completely executed FDX instances with exceptions, in hardware or operational model execution. Instructions may execute out-of-order across exception boundaries, requiring a modern definition for precision.

11.3 Relaxed behaviour of precise exceptions

Exceptions change the control flow and processor context, that is, the collection of system and special registers which control the execution of the machine. These include the current exception level (`PSTATE.EL`), masking of interrupts (`PSTATE.{D,A,I,F}`), processor flags, and so on. Changes to the context need not take effect immediately; to ensure that program-order-later instructions see such changes, exceptions are *context synchronising*: they ensure updates to the context are seen by later instructions. As a side-effect of that context synchronisation, exception boundaries impose some ordering.

We will see that the context synchronisation performed by the machinery is the primary mechanism that enforces order at the boundary of an exception. In addition to this, different classes of exceptions may come with their own additional ordering constraints: translation faults are bound by the constraints discussed in [Chapter 8](#), interrupts cannot happen before they are generated, and so on. However, we can set a baseline set of behaviours for exceptions by investigating the simplest kind of exception: the unencumbered exception-generating-instructions such as the SVC supervisor call. As such, throughout this section we will use exceptions from SVC instructions as an exploratory tool, but all behaviours described therein also apply to all other exception types.

In this section, we explain relaxed behaviour of precise exceptions through litmus testing. We start with the baseline out-of-order execution across exception boundaries (§11.3.1), before talking about context synchronisation in detail (§11.3.2). We continue with a collection of potentially interesting edge cases: changing privilege levels (§11.3.3), dependencies through exception machinery (§11.3.4), asynchronous exceptions (§11.3.5), then the stronger behaviour of specific types of exceptions (§11.3.6), before touching on how the instruction semantics needs to be adapted (§11.3.7), and finally we discuss a corner case when disabling context synchronisation (§11.3.8).

11.3.1 Out-of-order execution across exception boundaries

Before discussing the ordering that exception boundaries do impose, we will first see that, in general, exception boundaries do not act as memory barriers. Loads and stores may be executed out-of-order over an exception entry or an exception exit or the composition of both.

4599 Figure 11.6 contains a sample of shapes which show that the reads and writes are able to execute
 4600 out-of-order with respect to the various exception boundaries.

4601 As an example, the first of these, MP+dmb.sy+svc, is an MP (message passing) variant. The writer
 4602 thread is ordered by a full barrier. The reader thread is split in two: after the first read there is an SVC
 4603 which generates an exception and the exception handler performs the second read. We show this in the
 4604 execution diagrams with a single svc edge between the two reads.

4605 11.3.2 Context synchronisation and speculation

4606 Updates to the context, such as writes to system registers, need synchronisation to be guaranteed to have
 4607 an effect. We do not model the behaviour of such context-changing operations when such synchronisation
 4608 is not performed. Instead, we merely identify when and how exceptions are context-synchronising, and
 4609 note that this has a knock-on effect on memory accesses.

4610 Architecturally, a context synchronisation event guarantees that no instruction program-order-after the
 4611 event is observably fetched, decoded, or executed until the context-synchronising event has happened [68,
 4612 p. 14752, B2.10.1]. A simple microarchitectural implementation for context synchronisation is to flush the
 4613 pipeline: restarting all program-order-later instances once the context-synchronising effect occurs. More
 4614 complex implementations may be more clever, as long as they preserve the semantics.

4615 Software can explicitly generate context-synchronising events by issuing an Instruction Synchronisation
 4616 Barrier (ISB). Context synchronisation can also happen implicitly, for example on exception entry and
 4617 exit. This is the case in Arm, except in a rare use case we return to in §11.3.8.

4618 The effect of context synchronisation events in exception boundaries is that any instance after the boundary
 4619 has an ISB-equivalent dependency on the instances before the boundary. This mechanism implies the
 4620 following fundamental invariant: *context synchronising exception boundaries are never taken speculatively*.
 4621 This limits speculation of such boundaries to the same well-understood extent as speculation of ISBs. This
 4622 invariant has interesting interactions with external aborts, which we discuss in §11.4.

4623 The fact that context-synchronising exception boundaries cannot be taken speculatively implies that the
 4624 code inside an exception handler cannot execute before the exception entry's control-flow is determined
 4625 (see MP+dmb+ctrlsvc (Figure 11.7, p.190)); and similarly, cannot return before the ERET's control-flow is
 4626 determined (see MP+dmb+ctrleret (Figure 11.7, p.190)).

4627 11.3.3 Privilege level

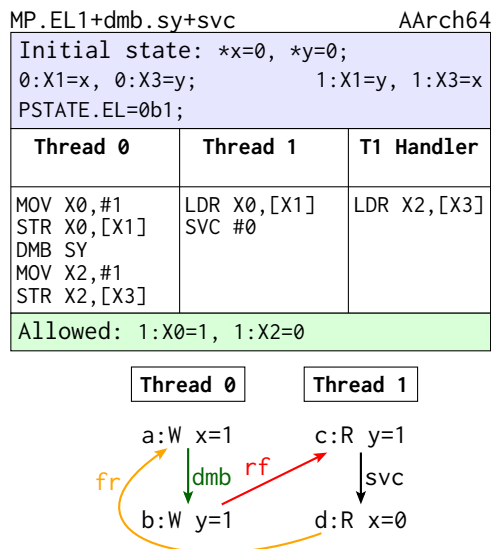
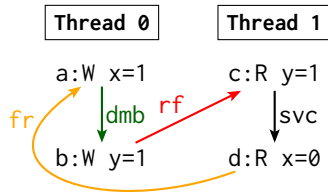


Figure 11.8: Same-exception-level exceptions are no stronger or weaker.

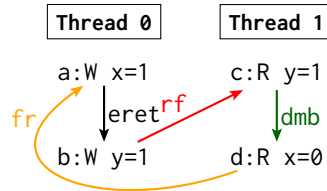
MP+dmb.sy+svc AArch64

Initial state: *x=0, *y=0; 0:X1=x, 0:X3=y; 1:X1=y, 1:X3=x		
Thread 0	Thread 1	T1 Handler
MOV X0,#1 STR X0,[X1] DMB SY MOV X2,#1 STR X2,[X3]	LDR X0,[X1] SVC #0	LDR X2,[X3]
Allowed: 1:X0=1, 1:X2=0		



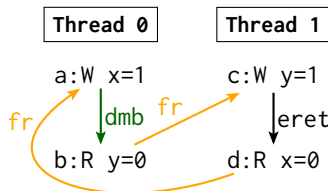
MP+eret+dmb.sy AArch64

Initial state: *x=0, *y=0; 0:X1=x, 0:X3=y; 1:X1=y, 1:X3=x		
Thread 0	T0 Handler	Thread 1
SVC #0 MOV X2,#1 STR X2,[X3]	MOV X0,#1 STR X0,[X1] ERET	LDR X0,[X1] DMB SY LDR X2,[X3]
Allowed: 1:X0=1, 1:X2=0		



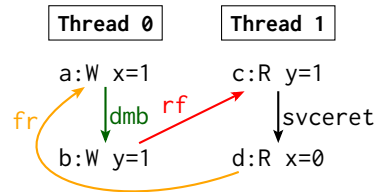
SB+dmb.sy+eret AArch64

Initial state: *x=0, *y=0; 0:X1=x, 0:X3=y; 1:X1=y, 1:X3=x		
Thread 0	Thread 1	T1 Handler
MOV X0,#1 STR X0,[X1] DMB SY LDR X2,[X3]	SVC #0 LDR X2,[X3]	MOV X0,#1 STR X0,[X1] ERET
Allowed: 0:X2=0, 1:X2=0		



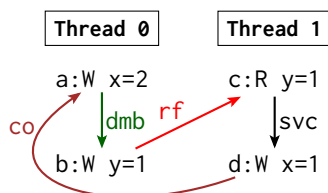
MP+dmb.sy+svceret AArch64

Initial state: *x=0, *y=0; 0:X1=x, 0:X3=y; 1:X1=y, 1:X3=x		
Thread 0	Thread 1	T1 Handler
MOV X0,#1 STR X0,[X1] DMB SY MOV X2,#1 STR X2,[X3]	LDR X0,[X1] SVC #0 LDR X2,[X3]	ERET
Allowed: 1:X0=1, 1:X2=0		



S+dmb.sy+svc AArch64

Initial state: *x=0, *y=0; 0:X1=x, 0:X3=y; 1:X1=y, 1:X3=x		
Thread 0	Thread 1	T1 Handler
MOV X0,#2 STR X0,[X1] DMB SY MOV X2,#1 STR X2,[X3]	LDR X0,[X1] SVC #0	MOV X2,#1 STR X2,[X3]
Allowed: 1:X0=1, *x=2		



LB+svc-erets AArch64

Initial state: *x=0, *y=0; 0:X1=x, 0:X3=y; 1:X1=y, 1:X3=x			
Thread 0	T0 Handler	Thread 2	T1 Handler
LDR X0,[X1] SVC #0 MOV X2,#1 STR X2,[X3]	ERET	LDR X0,[X1] SVC #0 MOV X2,#1 STR X2,[X3]	ERET
Allowed: 0:X0=1, 1:X0=1			

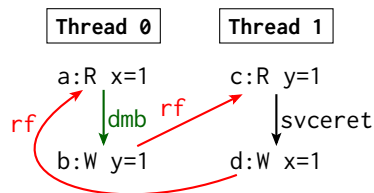


Figure 11.6: Reads and writes may be executed out-of-order across exception entry, exit, or even both.

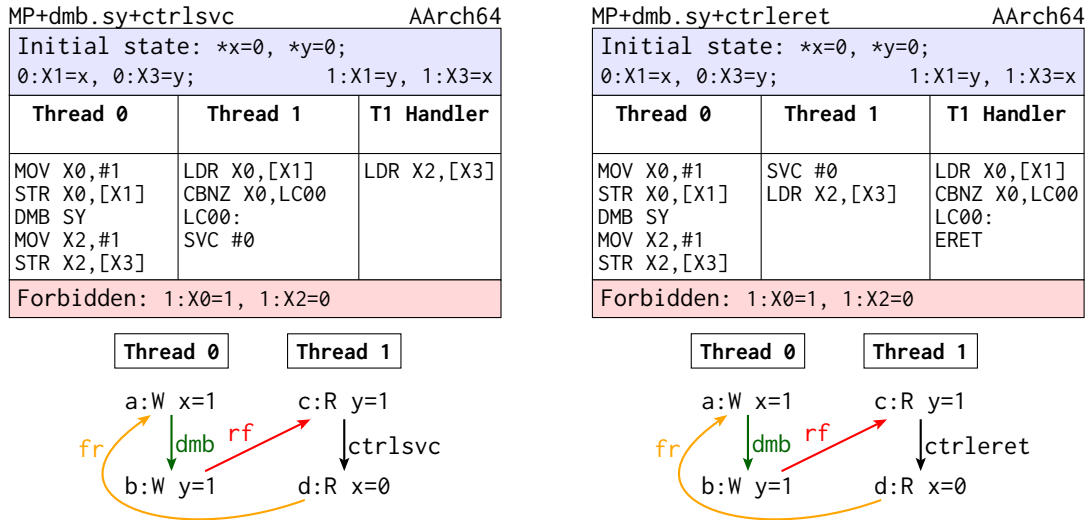


Figure 11.7: Context synchronising exception entry (and returns) are not executed speculatively.

4628 The privilege level (exception level) has little to no additional effect on the behaviours we present: their
 4629 allowed/forbidden status remains the same whether the privilege goes up/down in entry/exit or remains
 4630 the same. For example in the [MP.EL1+dmb+svc](#) test (Figure 11.8) the exception is taken *from* EL1 and
 4631 *to* EL1, but this does not affect any of the machinery (except which vector is used). As before, this is a
 4632 general statement about the exception machinery, and specific types of exceptions may have additional
 4633 constraints: e.g. translation faults cannot be caused by out-of-context translations, where the context
 4634 depends on the exception level (§8.8.1).

4635 **Store forwarding** It is permitted for writes to be forwarded from a store to a read across exception entry
 4636 and return. For example in the [SB+dmb+rfsvc-addr](#) test (Figure 11.9), the store in Thread 1 is observed
 4637 by the load in the exception handler (at a higher privilege level) ‘early’, before it is propagated globally.

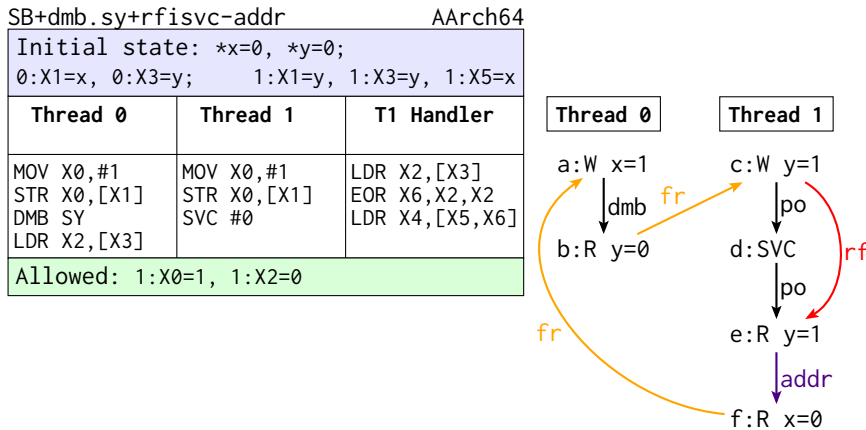


Figure 11.9: Forwarding into a non-speculative handler.

11.3.4 Dependency through system registers

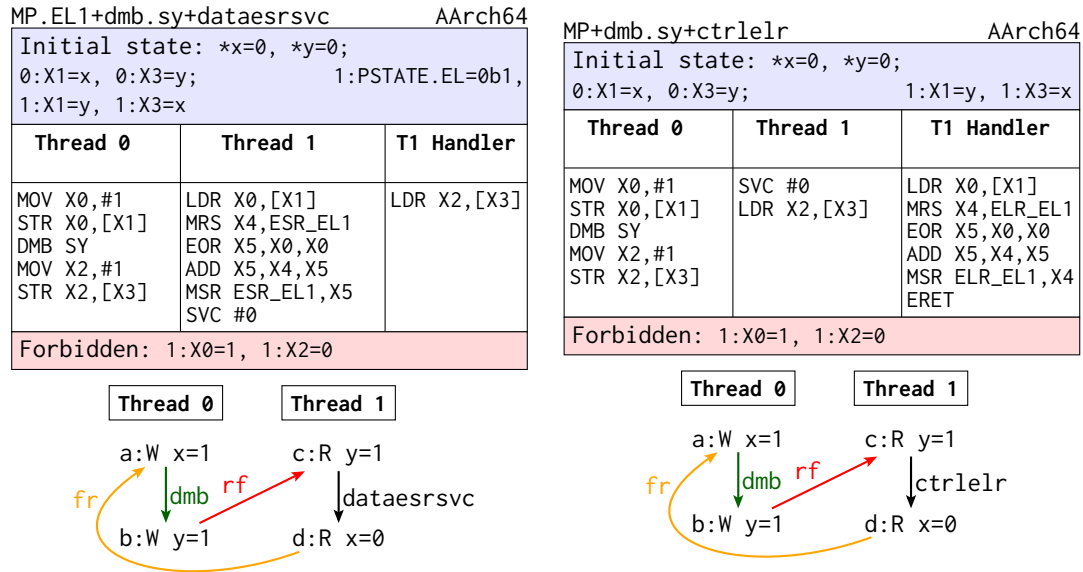


Figure 11.10: System registers and context synchronisation

Where exceptions are taken to and returned to are part of the context, and their respective registers must be read by the exception machinery on taking and returning from the exception. These registers are not read-only; software can write to them. Therefore, they can be involved in register dependency chains. While we do not attempt, in this work, to build a general model of dependencies, we touch on this particular aspect briefly.

Dependencies on system register accesses compose with ordering from context synchronisation events to program-order-later instructions. The `MP.EL1+dmb+dataesrsvc` test (Figure 11.10) demonstrates that a write to the system register ESR that depends on a read forbids reordering this read across the boundary, even though resolving the dependency does not affect the exception.

The ELR register is a *special-purpose register*, and is therefore ‘self-synchronising’, unlike system registers [81, D19.1.2, p6331]. Therefore, writes to the ELR do not need context synchronisation to guarantee that they are seen by program-order-later instructions, and this means that dependencies into the ELR are preserved automatically, for example, in the `MP+dmb+ctrlelr` test (Figure 11.10).

This has two related subtleties, and is currently under investigation by Arm. The Software Thread ID Register (TPIDR) is a system register in which the operating system can store thread identifying information, but has no relevant indirect effects. Further testing and discussions may clarify whether it forbids reordering. While dependencies through special-purpose registers are preserved, context synchronisation does not necessarily need to wait for those writes, and so these dependencies do not necessarily pass to instructions after context synchronisation (in contrast to system register writes).

11.3.5 Ordering from asynchronous exceptions

Asynchronous exceptions cannot be taken speculatively. Therefore, all instructions program-order-after an asynchronous exception happen after that exception.

11.3.6 Exception-specific mechanisms

Some exceptions on some implementations involve additional mechanisms. For example, when an implementation supports Enhanced Translation Synchronisation the translation-table-walks which generate MMU faults gain additional ordering from program-order-previous instances, see §8.4.3. Figure 11.11 compares a message-passing shape involving a translation fault with an asynchronous interrupt.

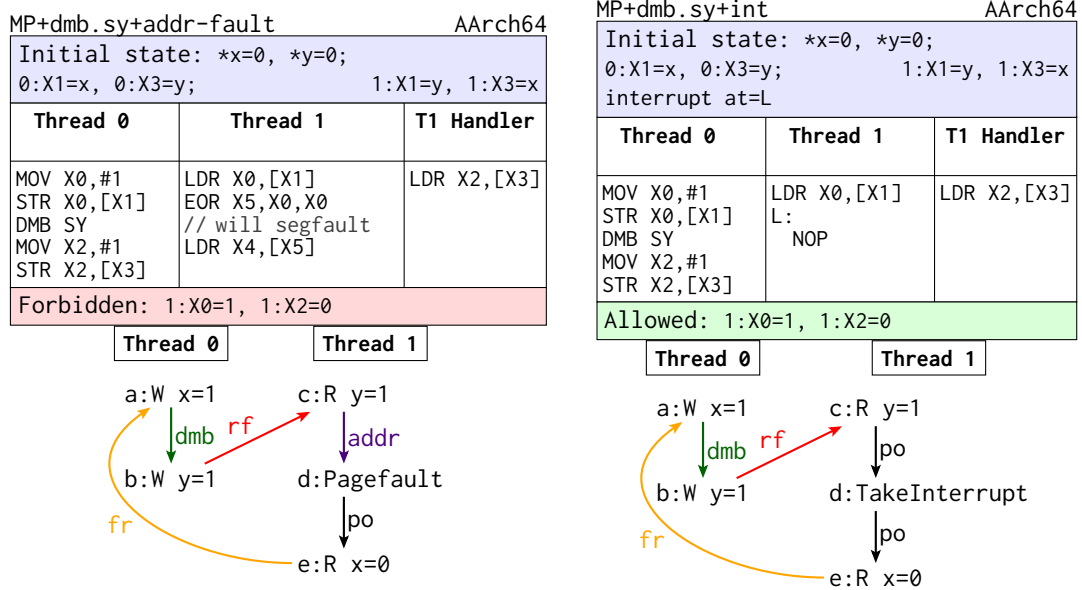


Figure 11.11: Different exception kinds can have different behaviour.

11.3.7 Exceptions and the intra-instruction semantics

Wherever possible, we want to interpret the intra-instruction ASL ordering as preserved, for conceptual simplicity, memory-model tool execution, and reasoning. This has previously been possible except in a few specific cases that are inherently concurrent: instructions that do multiple accesses, and CSEL, CAS, SWAP, etc. Exceptions introduce a new interesting case for instructions that do a register writeback concurrently with a memory access. For example, STR (immediate) has ‘Post-index’ and ‘Pre-index’ versions [81, C6.2.322, p1996]. The post-index STR Xt, [Xn], #8, for example, stores the value in Xt to the address initially in register Xn and increments Xn by 8. The Arm ARM ASL for STR puts that register write at the end, after the memory access has completed.

The architectural intent is that program-order-later instances that depend on Xn can go ahead early, e.g. before the data in register Xt is available to be written to memory. The related litmus tests have previously been observed on hardware [119].

Previous work captured this allowed by having the register writeback before the memory access in the instruction semantics. However, exceptions require more care: when the memory access generates an exception, the writeback register should appear unchanged to instances after the exception boundary.

Currently, the Armv9.4 ASL, and therefore the Sail derived from it, does not account for the relaxed behaviour of exceptions around these register writebacks. There are a number of proposed solutions, ranging from including code to cleanup after exceptions in the ASL, to treating register writes fully relaxed, like memory writes: with speculative ‘forwarding’ and discarding on an exception. As was discussed earlier we will need some similar change for relaxed system registers (§8.7.1), and for intra-instruction parallelism (§8.4.9). No solution has yet been implemented, and so the model here does not account for these cases.

11.3.8 Disabling context synchronisation

So far we have assumed exception boundaries are context synchronising. However, Arm has an optional feature, FEAT_ExS, which provides two new fields, EIS and EOS, in the SCTLR_ELx system control register. These allow software to disable context synchronisation on exception entry and return, respectively. While the semantics seems clear for these systems, the programming model is unpredictable and hard to program correctly, and so this configuration is rarely encountered in practice.

The result of switching off context synchronisation on exception boundaries is to weaken the previously described speculation tests: permitting speculation of the entry or exit of non-context-synchronising exception boundaries, and all the behaviours associated thereof.

4696 11.4 Synchronous external aborts

4697 The memory system may detect errors such as data corruption independently of the MMU or Debug
4698 hardware, e.g. using parity bits or error correcting code. In those cases, it will signal the error by a
4699 class of exceptions called *external aborts*. The architecture does not define when implementations report
4700 such aborts synchronously [97, D7.6, I_{PJHZS}]. It is implementation defined whether an external abort is
4701 reported as a synchronous external abort (under the ‘Data abort’ class) or asynchronously as a system
4702 error, and what errors are reported as external aborts at all [97, R_{FNVVJ}].

4703 Instances program-order-after a potential cause for synchronous external aborts are considered speculative
4704 until any such synchronous external abort can be ruled out. This results in stronger behaviour (§11.4.1).
4705 In an implementation that always reports external aborts asynchronously, the later instances become
4706 non-speculative earlier, allowing them to exhibit weaker behaviours.

4707 In general, systems want to report errors as synchronously as possible. When errors are reported
4708 asynchronously, in general, the only recovery is to wind down the aborting process. The Arm Reliability,
4709 Availability, and Serviceability (RAS) extension adds some ability for more fine-grained recovery procedures,
4710 but this extension is a substantial component of the architecture, far beyond the scope of this work.

4711 11.4.1 Behaviour resulting from synchronous external aborts

4712 There is an asymmetry between reads and writes with respect to speculation: reads can be satisfied
4713 speculatively, whereas writes cannot be propagated speculatively. We must therefore consider reads and
4714 writes separately.

4715 Instructions program-order-after stores are executing speculatively up until the store has reached the point
4716 in its execution where no further synchronous exceptions can occur. If a store may generate a synchronous
4717 external abort, then that point is (at least) once it has propagated to memory. This means that events
4718 program-order-after an unpropagated store must also be considered to be speculative until the store has
4719 propagated. For example, this forbids out-of-order propagation of writes (e.g. as in MP+po+addr).

4720 Reads program-order-after writes are permitted to execute early, even speculatively, unless otherwise
4721 forbidden by some other part of the model, e.g. with an interposing context-synchronisation-event. For
4722 example, when there is the possibility of synchronous external aborts on a store, reads program-order-after
4723 ISBs program-order-after the store are forbidden from satisfying before the propagation of that store (e.g.
4724 as in R+dmb.sy+isb).

4725 Perhaps more interestingly, if a load may generate a synchronous external abort, then program-order-later
4726 instances are speculative until the load has completed all its reads, and is non-restartable. This means that
4727 writes program-order-after that read are forbidden from executing out-of-order. This forbids interesting
4728 tests which would otherwise be allowed, namely load-buffering (LB+pos) and MP with a plain ISB after
4729 one load (MP+dmb.sy+isb) [120].

4730 **Load buffering and the out-of-thin-air problem** This has an important and hitherto not well-understood
4731 impact on programming-language concurrency models. Ruling out LB enables substantially simpler design
4732 of programming language concurrency models: they can execute instructions in-order and merely keep a
4733 history of the writes seen so far, e.g. [121], and thereby avoid the notorious out-of-thin-air problem [122].
4734 These simpler semantics support a line of model checkers for C/C++ and LLVM [123, 124, 125]. In
4735 contrast, the presence of LB seems to require significant sophistication [126, 122, 127, 128, 129, 42, 28, 130].

An axiomatic model for precise exceptions

We now give a formal semantics that describes the concurrent behaviour of precise exceptions on Arm-A. We give it as an extension of the previous model of Deacon et al. [7], as was recalled in Chapter 2. The full model extended with exceptions is given in Figure 12.1.

The model is parameterised along two axes:

- ▷ FEAT_ExS corresponds to the feature of the same name being implemented; we do not support runtime changes of the related SCTLR_ELx.{EIS,EOS} fields, and so fix them as variants.
- ▷ SEA_R and SEA_W correspond to the implementation-defined choice of whether loads or stores may generate synchronous external aborts.

Most current hardware does not support FEAT_ExS, and moreover, we expect that most software would not use it. However, its semantics is relatively straight-forward as we understand it, and so we include it in our model.

The SEA variants in this model are not architecturally-defined identifiers. In fact, in the absence of actually observing a fault directly there appears no architectural way to identify the choice beyond running the litmus tests presented in Chapter 11. These two variants capture whether *any* store or load respectively, *could* generate a synchronous external abort, even though the model does not consider executions in which such aborts actually occur.

12.1 Extended candidates

To support precise exceptions, we add new events to the candidate execution:

- ▷ TE (take exception), and TakeInterrupt, and ERET (exception return). These correspond to the synchronisation points (whether or not they are synchronising) of taking or returning from an exception.
- ▷ MRS and MSR events for the reading and writing (respectively) of system registers, corresponding to the identically-named Arm instructions.

Exceptions and program-order Program-order includes all the events of the thread, even with interposing exceptions. That is, program-order is not discontinuous, at least for precise exceptions. We therefore include all the new events in program-order. This includes the events from instructions directly before and after taking or returning from an exception.

Interrupts While we do not model inter-processor interrupts or the generic interrupt controller, we do support precise asynchronous exceptions generally (e.g. timers).

Candidates can, at any point in thread, have an instance which does not follow from the natural intra-instruction semantics, but corresponds to pending an interrupt, i.e. setting the appropriate bit in the ISR. The intra-instruction semantics then will take the interrupt at the appropriate time.

For performance reasons in the executable-as-a-test-oracle implementation within `isla-axiomatic` we do not allow arbitrary interrupts, see §13.2.

```

1 "Arm-A exceptions"
2
3 include "cos.cat"
4 include "arm-common.cat"
5
6 (* might-be speculatively
   executed *)
7 let speculative =
8   ctrl
9   | addr; po
10  | if "SEA_R"
11    then [R]; po
12    else 0
13  | if "SEA_W"
14    then [W]; po
15    else 0
16
17 (* context-sync-events *)
18 let CSE =
19   ISB
20   | if "FEAT_ExS" & ~"EIS"
21     then 0
22     else TE
23   | if "FEAT_ExS" & ~"EOS"
24     then 0
25     else ERET
26
27 let ASYNC =
28   TakeInterrupt
29
30 (* observed by *)
31 let obs = rfe | fr | co
32
33 (* dependency-ordered-before *)
34 let dob =
35   addr | data
36   | speculative ; [W]
37   | speculative ; [ISB]
38   | (addr | data); rfi
39
40
41
42 (* atomic-ordered-before *)
43 let aob =
44   rmw
45   | [range(rmw)]; rfi; [A|Q]
46
47 (* barrier-ordered-before *)
48 let bob =
49   [R] ; po ; [dmbld]
50   | [W] ; po ; [dmbst]
51   | [dmbst]; po; [W]
52   | [dmbld]; po; [R|W]
53   | [L]; po; [A]
54   | [A | Q]; po; [R | W]
55   | [R | W]; po; [L]
56   | [dsb]; po
57
58 (* contextually-ordered-before *)
59 let ctxob =
60   speculative; [MSR|CSE]
61   | [MSR]; po; [CSE]
62   | [CSE]; po
63
64 (* async-ordered-before *)
65 let asyncob =
66   speculative; [ASYNC]
67   | [ASYNC]; po
68
69 (* Ordered-before *)
70 let ob = (obs | dob | aob |
71   bob | ctxob | asyncob)+
72
73 (* Internal visibility
   requirement *)
74 acyclic po-loc | fr | co | rf as
   internal
75
76 (* External visibility
   requirement *)
77 irreflexive ob as external
78
79 (* Atomic: Basic LDXR/STXR
   constraint to forbid
   intervening writes. *)
80 empty rmw & (fre; coe) as atomic

```

Figure 12.1: Arm-A exceptional model (grayed out parts are unchanged from the original model).

12.2 Extended relations

We expand ordered-before:

- ▷ Wherever `ctrl1|(addr;po)` was used before, we also include instructions program-order-after reads or writes when in the relevant SEA variant. With those variants, the instructions program-order-after those events are speculative up until the memory access has completed.
- ▷ The previous model's use of ISB was purely for its context synchronisation effect. Accordingly, wherever `[ISB]` was used before, we include exception entry (TE) and exit (ERET), unless we are in the variant where context synchronisation on those events is disabled.
- ▷ We extend barrier-ordered-before with the DSB barriers. The barrier event classes are upwards-closed, so that `DSB.SY` is included in all the `dmb` events.
- ▷ We add a context-ordered-before (`ctxob`) sub-clause to the ordered-before relation, which captures the ordering of context-changing operations and context-synchronisation: namely, that context-changes and context-synchronisation cannot happen speculatively; that all context-changes are ordered before any context-synchronisation; and that no instruction program-order-after context-synchronisation can be executed until the synchronisation is complete.
- ▷ We add an async-ordered-before (`asynco`) clause to ordered-before, capturing that asynchronous events (such as interrupts) cannot be done speculatively, and instructions program-order-after them may not happen before the asynchronous event which precipitated them.

12.3 Challenges in defining precision

The phenomena we described in §11.3 highlight how the historical definition of precision does not account for relaxed memory. The problem is then *how to adequately define precision in a relaxed-memory setting*. This challenge is hinted at in the way the Arm reference manual [81, D1.3.1, p5355] defines precision as:

An exception is *precise* if on taking the exception, the hardware thread (aka processing element, PE) state and the memory system state is consistent with the PE having executed all of the instructions up to but not including the point in the instruction stream where the exception was taken from, and none afterwards. [except that in certain specific cases some registers and memory values may be UNKNOWN]

This definition explicitly allows various side effects of an instruction executing when an exception is taken to be visible. The details are intricate, but in outline: registers that would be written by the instruction but which are not used by it (to compute memory access addresses) can become UNKNOWN, and for instructions that involve multiple single-copy-atomic memory writes (e.g. misaligned writes and store-pair instructions), where each write might generate an exception (e.g. a translation fault), the memory locations of the writes that do not generate exceptions become UNKNOWN. These side effects could be observed by the exception handler, and the memory write side effects could be observed by other threads doing racy reads. Hardware updates to page-table access flags and dirty bits, and to performance counters, could also be observable. This means that the abstraction of a stream of instructions executed up to a given point does not account for the relaxed-memory behaviour.

Arm *classify* particular kinds of exceptions as precise or not, but all the above makes it hard to *define* in general what it means for an exception to be precise in a relaxed setting.

The ultimate architectural intent of precision is that it is sufficient to meaningfully resume execution after the exception. For example, for software that does mapping on demand, when an instruction causes a fault by accessing an address which is not currently mapped, the exception handler will map that address and return. This means that re-executing the original instruction will overwrite these UNKNOWNs, and will have ordering properties much like the original instruction would have had if the mapping had already been in place.

Our models are complete enough to reason about such cases in concrete examples. However, a general definition of precision, and the accompanying reasoning principle, would have to capture assumptions

4815 about the exception handler and its concurrent context to ensure that they do not observe the above
4816 side effects. More straightforwardly, the above definition of what becomes UNKNOWN would have to be
4817 codified, as that is not currently in the ASL architectural pseudocode.

4818 Exceptions may also be *imprecise*, in which case the behaviour is very loosely constrained, and the current
4819 architecture does not give well-defined guarantees in the presence of imprecise exceptions.

4820 **12.4 Scope and limitations**

4821 We do not give semantics to imprecise exceptions. It is unclear how to do so at an architectural level.

4822 We do not define the behaviour of ‘constrained unpredictable’, and merely flag when it is triggered.
4823 Clarifying it will require substantial extensive discussions with Arm architects, likely affecting the wording
4824 in the architectural specifications, beyond the scope of this work. We do not model switching between
4825 Arm FEAT_ExS modes (§11.3.8): they are supported architecturally, but are not commonly implemented.
4826 Finally, while we believe our models correctly capture the Arm architectural intent, and that it gives a
4827 solid basis for programmers, this is not an authoritative definition of the architecture, and is subject to
4828 change.

Validating the exceptions model

13.1 Validating against hardware

We extend the harness described in [Chapter 10](#), and run a set of 55 hand-written tests on a small collection of devices: Raspberry Pi 3B+, 4B, and 5; an ODROID N2+ with an Amlogic S922X SoC; and an Apple Mac Mini with Apple M2 silicon SoC. The results from that testing can be found in [Table 13.1](#).

We note that many devices do not observably re-order loads with respect to writes or context-synchronisation: of our devices, only the ODROID's S922X SoC had a core (its Arm Cortex-A73) which exhibit those behaviours on usermode tests. For that reason, we did not expect, except on that core, to observe re-ordering loads across exception boundaries.

We find that out-of-order execution of stores across exception boundaries is readily observable over the full range of devices. The Apple M2 is the only device which did not exhibit store-store re-ordering across an exception boundary, but it does not exhibit normal store-store re-ordering across context-synchronisation in usermode tests, so this is not unexpected.

As such, the hardware results for exceptions are disappointingly weak: the hardware we have access to is not aggressive enough in the base usermode behaviours to be expecting to observe the behaviours we seek here, and the number of runs is not substantial enough to draw any inferences on the envelope of behaviours they implement. Further hardware testing will be required to build the necessary confidence, especially around FEAT_ExS and the differences in the exception kinds.

13.2 Executable-as-a-test-oracle implementation

We implement the model as an executable-as-a-test-oracle implementation in Isla [\[46\]](#),

To support tests with asynchronous exceptions, we added a construct to specify a label where the exception will occur, so that Isla then pends an interrupt at that program point.

The instruction semantics we use is a translation into the Sail language of the Armv9.4-A ASL specification, including the top-level function provided by Arm [\[112\]](#). The translation process [\[45\]](#) is mostly automatic, requiring select manual interventions mostly due to differences in the type systems of ASL and Sail. We also added patches to support the integration with Isla, in particular adding hooks to expose information about exceptions being taken in a form that can be readily consumed by Isla. In doing so, we encountered and fixed some bugs in the ASL model related to uses of uninitialised fields in data structures, as well as missing checks for implemented processor features that led to spurious system register accesses.

The results from the model over each of the variants can be found in [Table 13.2](#).

Table 13.1: Exceptions hardware refs. Columns are, respectively, an ODROID N2+ (Amlogic S99X, ‘big’ cores only, Arm Cortex-A73 r0p2), an Apple M2, and Raspberry Pis 3B+ (Arm Cortex-A53 r0p4), 4B (Arm Cortex-A72 r0p3), and 5 (Arm Cortex-A76 r1p4).

Name	s922x	m2	rpi3b+	rpi4b	rpi5
LB+svc-dmb-erets	0/18M	0/360M	0/1M	0/19M	0/11M
LB+svc-erets	0/18M	0/360M	0/1M	0/19M	0/11M
LB+svcs	388/18M	0/360M	0/1M	0/19M	0/11M
MP+dmb+ctrl-eret	0/18M	0/360M	0/1M	0/19M	0/11M
MP+dmb+ctrl-rfisvceret-addr	0/18M	0/360M	0/22M	0/108M	0/39M
MP+dmb+ctrl-svc	0/18M	0/360M	0/1M	0/19M	0/11M
MP+dmb+ctrl-eret	0/18M	0/360M	0/22M	0/108M	0/39M
MP+dmb+data-svc	0/18M	0/360M	0/1M	0/19M	0/11M
MP+dmb+dmb-eret	0/18M	0/360M	0/1M	0/19M	0/11M
MP+dmb+eret	0/18M	0/360M	0/1M	0/19M	0/11M
MP+dmb+eret-dmb	0/18M	0/360M	0/1M	0/19M	0/11M
MP+dmb+eret=addr	0/18M	0/0	0/1M	0/19M	0/11M
MP+dmb+svc	84/18M	0/360M	0/1M	0/19M	0/11M
MP+dmb+svc-addreret	0/18M	0/360M	0/1M	0/19M	0/11M
MP+dmb+svc-dmb	0/18M	0/360M	0/1M	0/19M	0/11M
MP+dmb+svc-dmb-eret	0/18M	0/360M	0/1M	0/19M	0/11M
MP+dmb+svc-eret	0/18M	0/360M	0/33M	0/19M	0/11M
MP+dmb+svcnocis	23/18M	0/360M	0/1M	0/19M	0/11M
MP+eret+addr	22K/18M	0/360M	63/1M	0/19M	2/11M
MP+eret+dmb	18K/18M	0/360M	1K/33M	262/19M	20/11M
MP+eret+svc	9K/18M	0/360M	244/21M	256K/107M	20/39M
MP+erets	29K/18M	0/360M	30/1M	59/19M	16/11M
MP+svc+addr	17K/18M	0/360M	59/1M	0/19M	8/11M
MP+svc+dmb	18K/17M	0/360M	80/1M	3/19M	876/11M
MP+svc+eret	22K/17M	0/360M	1K/21M	33/107M	77/39M
MP+svc-W-eret-W+addr	14K/13M	0/0	0/0	0/16M	1/6M
MP+svc-dmb+addr	0/17M	0/360M	0/1M	0/19M	0/11M
MP+svc-dmb-eret+addr	0/17M	0/360M	0/1M	0/19M	0/11M
MP+svc-eret+addr	13K/17M	0/360M	52/1M	0/19M	2/11M
MP+svc-erets	3K/17M	0/360M	42/1M	2/19M	8/11M
MP+svcs	8K/17M	0/360M	31/1M	0/19M	20/11M
MP.EL1+dmb+ctrlvbarsvc	0/17M	0/360M	0/21M	0/108M	0/39M
MP.EL1+dmb+svc	29/17M	0/360M	0/33M	0/12M	0/11M
S+dmb+eret	0/17M	0/360M	0/33M	0/12M	0/11M
S+dmb+svc	0/17M	0/360M	0/33M	0/12M	0/11M
S+erets	0/17M	0/360M	0/1M	0/19M	0/11M
S+svc-dmb-erets	0/17M	0/359M	0/1M	0/19M	0/11M
S+svc-erets	0/17M	0/359M	0/1M	0/19M	0/11M
S+svcs	0/17M	0/359M	0/1M	0/19M	0/11M
SB+dmb+eret	38/17M	12K/359M	162K/33M	85K/12M	2K/11M
SB+dmb+rfi-ctrl-eret	17K/17M	10K/359M	10K/1M	42K/19M	46/11M
SB+dmb+rfi-ctrl-svc	13K/17M	8/359M	15K/1M	2K/18M	58K/11M
SB+dmb+rfi-eret-addr	16K/16M	6K/359M	591K/21M	8K/107M	54/39M
SB+dmb+rfisvc-addr	18K/16M	12/359M	839K/21M	2K/106M	135K/39M
SB+dmb+svc	195/16M	14/359M	351K/33M	458/11M	63K/11M
SB+svc-dmb-erets	0/16M	0/359M	0/1M	0/18M	0/11M
SB+svc-erets	9K/16M	0/359M	22K/1M	7K/18M	38/11M
SB+svcs	2K/16M	0/359M	534K/21M	0/106M	646K/39M
SEA_R_detect	0/16M	359M/359M	0/1M	0/18M	0/10M
SEA_W_detect	0/16M	0/359M	0/1M	0/18M	0/10M
MP+dmb+eret-svc	0/4M	0/360M	0/1M	0/3M	0/5M
MP.EL1+dmb+eret	0/4M	0/360M	0/1M	0/3M	0/5M
MP.EL1+dmb+eret-svc	0/4M	0/360M	0/1M	0/3M	0/5M
MP.EL1+dmb+svc-eret	0/4M	0/360M	0/1M	0/3M	0/5M
SB.EL1+erets	15K/3M	0/359M	25K/1M	148/2M	43/5M
SB.EL1+svc-erets	3K/3M	0/359M	19K/1M	1K/2M	17/5M

Table 13.2: Exceptions model refs

Name	No features	FEAT_ExS	SEA_R	SEA_W	SEA_R&W
LB+svc-dmb-erets	forbid	forbid	forbid	forbid	forbid
LB+svc-erets	allow	allow	forbid	allow	forbid
LB+svcs	allow	allow	forbid	allow	forbid
MP+daifset+dmb	allow	allow	allow	forbid	forbid
MP+dmb+ctrl-eret	forbid	allow	forbid	forbid	forbid
MP+dmb+ctrl-rfisvceret-addr	forbid	allow	forbid	forbid	forbid
MP+dmb+ctrl-svc	forbid	allow	forbid	forbid	forbid
MP+dmb+ctrl-lr	forbid	allow	forbid	forbid	forbid
MP+dmb+daifset	allow	allow	allow	allow	allow
MP+dmb+dmb-eret	forbid	forbid	forbid	forbid	forbid
MP+dmb+eret-dmb	forbid	forbid	forbid	forbid	forbid
MP+dmb+eret-svc	allow	allow	forbid	allow	forbid
MP+dmb+eret	allow	allow	forbid	allow	forbid
MP+dmb+eret=addr	forbid	forbid	forbid	forbid	forbid
MP+dmb+svc-addreret	allow	allow	forbid	allow	forbid
MP+dmb+svc-dmb-eret	forbid	forbid	forbid	forbid	forbid
MP+dmb+svc-dmb	forbid	forbid	forbid	forbid	forbid
MP+dmb+svc-eret	allow	allow	forbid	allow	forbid
MP+dmb+svc	allow	allow	forbid	allow	forbid
MP+dmb+svcnoeis	allow	allow	forbid	allow	forbid
MP+eret+addr	allow	allow	allow	forbid	forbid
MP+eret+dmb	allow	allow	allow	forbid	forbid
MP+eret+svc	allow	allow	allow	allow	forbid
MP+erets	allow	allow	allow	allow	forbid
MP+svc+addr	allow	allow	allow	forbid	forbid
MP+svc+dmb	allow	allow	allow	forbid	forbid
MP+svc+eret	allow	allow	allow	allow	forbid
MP+svc-dmb+addr	forbid	forbid	forbid	forbid	forbid
MP+svc-dmb-eret+addr	forbid	forbid	forbid	forbid	forbid
MP+svc-eret+addr	allow	allow	allow	forbid	forbid
MP+svc-erets	allow	allow	allow	allow	forbid
MP+svcs	allow	allow	allow	allow	forbid
MP.EL1+dmb+ctrlvbarsvc	forbid	allow	forbid	forbid	forbid
MP.EL1+dmb+eret-svc	allow	allow	forbid	allow	forbid
MP.EL1+dmb+eret	allow	allow	forbid	allow	forbid
MP.EL1+dmb+svc-eret	allow	allow	forbid	allow	forbid
MP.EL1+dmb+svc	forbid	forbid	forbid	forbid	forbid
S+dmb+eret	allow	allow	forbid	allow	forbid
S+dmb+svc	allow	allow	forbid	allow	forbid
S+erets	allow	allow	allow	allow	forbid
S+svc-dmb-erets	forbid	forbid	forbid	forbid	forbid
S+svc-erets	allow	allow	allow	allow	forbid
S+svcs	allow	allow	allow	allow	forbid
SB+daifsets	allow	allow	allow	allow	allow
SB+dmb+eret	allow	allow	allow	forbid	forbid
SB+dmb+rfi-ctrl-eret	allow	allow	allow	forbid	forbid
SB+dmb+rfi-ctrl-svc	allow	allow	allow	forbid	forbid
SB+dmb+rfi-eret-addr	allow	allow	allow	forbid	forbid
SB+dmb+rfisvc-addr	allow	allow	allow	forbid	forbid
SB+dmb+svc	allow	allow	allow	forbid	forbid
SB+svc+dmb-erets	forbid	forbid	forbid	forbid	forbid
SB+svcs	allow	allow	allow	forbid	forbid
SB.EL1+erets	allow	allow	allow	forbid	forbid
SB.EL1+svc-erets	allow	allow	allow	forbid	forbid
MP+dmb+ctrl-int	forbid	forbid	forbid	forbid	forbid
MP+dmb+int	allow	allow	allow	allow	allow
MP+int+dmb	allow	allow	allow	allow	allow

Conclusion

We presented models for three key parts of the Arm architecture required for systems software: instruction fetch and its required cache maintenance instructions; virtual memory and its required TLB maintenance instructions; and the baseline behaviour for precise exceptions. We have produced a corpus of hand-written litmus tests for these architectural aspects, covering a range of interesting hardware optimisations and software requirements. We have clarified the architectural intent for those tests, and produced models that capture that intent, as we understand it at the time of writing.

We produced axiomatic-style declarative semantics, in the standard cat language, for all three aspects of the architecture. Additionally we produced a microarchitectural-style operational semantics for the instruction fetch fragment intended equivalent to the axiomatic one.

We validated these models against a variety of hardware implementations, even finding some places where modern microprocessors deviate from the desired architectural intent. For instruction fetch, with Maranget we extended the herdttools suite to be able to generate new litmus tests, and ran those tests on hardware. We built a brand new test harness, `system-litmus-harness`, able to run tests on a variety of hardware at EL1, either bare metal or in KVM. We used this harness to produce experimental data for both the virtual memory and exceptions parts.

We made these models executable as a test oracle, allowing the user to experimentally check behaviours manually, or even do rudimentary model checking of a larger software pattern, by implementing them in the `isla-axiomatic` or `RMEM` tools. This allowed us to validate the models against each other where applicable, and against the architectural intent, and comparing the results from hardware test runs against the model's predictions.

Finally, for virtual memory, we proved a simple virtual memory abstraction which gives confidence that the model correctly captures a key property that the model is intended to have.

14.1 Limitations

While we endeavour to be as faithful to the architectural intent as we can, and to produce models that are sound abstractions of that intent, we have had to make tradeoffs in places.

We presented three models for three separate parts of the architecture, but did not merge them together into a single architectural model. The models can be unioned together to produce a combined model with all the events and relations from the models, but more work is needed to understand the interactions between the architectural features: instruction fetches are memory reads which themselves are translated, but where that translation behaves subtly different from the normal translations with different caching rules; translation and instruction fetch can both cause exceptions to happen; exceptions cause the control-flow to change and new instructions to be fetched; and so on. We do not imagine this is a particularly arduous or complex task, but one that we have not yet done.

We produced two new separate languages for defining litmus tests. Ideally, we would have one unified language that all tools (litmus, isla-axiomatic, and system-litmus-harness) all accept. As stated earlier, we do not believe there is a fundamental restriction to unifying these languages, as currently they have not diverged so far as to be incompatible.

4899 There are some places where it has become known that models presented in this work do not faithfully
4900 capture the architectural intent as it is known today. In particular around the reachability of pagetable
4901 entries, and invalidation of non-last-level pagetable entries, as was discussed earlier.

4902 **14.2 Future work**

4903 There are many areas where the work presented here is only the start, and where further effort could bear
4904 fruit.

4905 For more confidence in the architectural intent, more hardware testing (especially for the virtual memory
4906 tests) is essential. In particular, running at EL2 (for stage 2 tests), and over a more varied collection of
4907 devices.

4908 Capturing more of the systems architecture is always desirable. We made a start here, but this is no
4909 means the end. Modern systems software relies on much more of the architecture than just covered
4910 here, such as: the Arm generic interrupt controller, and virtualisation of interrupts; the variety of Arm
4911 features and extensions for virtual memory e.g. FEAT_ETS2, FEAT_BBM, FEAT_nTLBPA, access permissions,
4912 and cacheability, and shareability domains; device memory and DMA; and much more.

4913 With the models themselves, they can always be improved to be executable more efficiently, and the tools
4914 easier to use. `isla-axiomatic` can run the virtual memory tests, but needs optimisations to be able to
4915 run in any reasonable timeframe, and even then still takes hours on a modern high-end machine. This
4916 seriously restricts the current usefulness of such tools to the average programmer.

4917 There are now many concurrently existing models for Arm, covering overlapping sets of features: the
4918 usermode model, which we recalled in [Chapter 2](#); for transactional memory, Chong et al. [131]; for
4919 persistent memory, Raad et al. [89]; and the official Arm model which contains an updated usermode,
4920 memory tagging, ifetch, and some overlapping parts of the VMSA, Alglave et al. [42, 97, 107]. Simply
4921 gluing these together into a single model would not be sound, as their interactions would need to be
4922 explored, and the architectural intent clarified first. However, it seems necessary for such work to be
4923 carried out to enable future verification efforts of complex systems.

4924 Work on relaxed systems, either on virtual memory, instruction fetching, or exceptions has not ceased at
4925 the finalization of this work. We are continuing to improve all the models given here, to engage in fruitful
4926 discussions with Arm, to produce new models for more of the architecture, and to build more confidence
4927 in the models we have already created.

4928 Hopefully, this work enables future work by researchers, academics, engineers, architects, and hardware
4929 designers, to better understand the environment as it is today. This hopefully will help them to produce
4930 clearer and more robust architectures, and to take the first steps in verifying the complex systems software
4931 that underpins so much of the modern base of computing with respect to the reality of the hardware we
4932 run them on.

Pocket guide to the Arm ISA

The litmus tests, as found in this thesis, use a relatively small subset of the whole ISA.

Refer to the Arm Architecture Reference Manual, Section C6 (‘A64 Base Instruction Descriptions’) for a more complete explanation of all the instructions.

A.1 Architectural concepts

Some terminology:

- ▷ AArch64 is the 64-bit execution mode.
- ▷ A64 is the name of the 64-bit ISA which AArch64 executes.
- ▷ PE (‘Processing Element’) is generic Arm terminology for a hardware thread/core.
- ▷ GPR (‘General-purpose register’) is one of the 31 ‘general-purpose’ registers.
- ▷ Immediate values are literal numeric values used in the instructions, as opposed to being read from registers.

Exception levels Arm execution is split into privilege levels (called *exception levels* in Arm), labelled from EL0 (least privileged execution) to EL3 (most privileged), see Fig A.1.

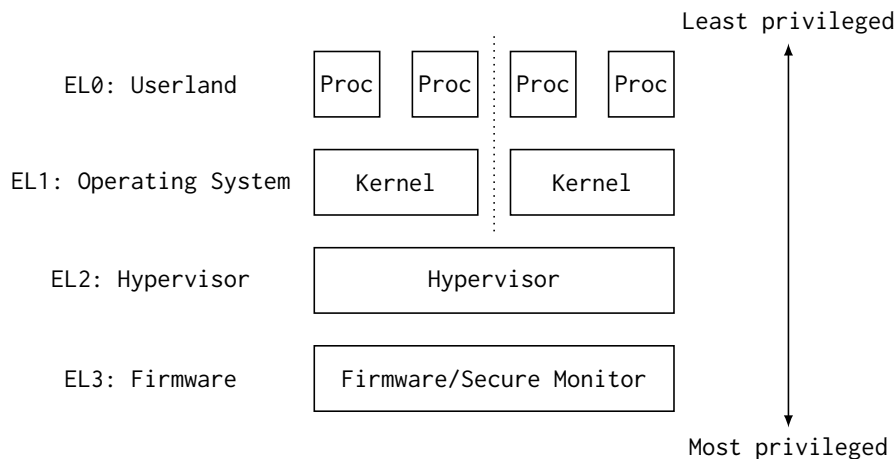


Figure A.1: Arm-A exception levels.

Registers A64 has:

- ▷ 31 general-purpose registers, named R0–R30.

- 4950 – R_n is an internal name, the register should be accessed via one of its aliases: X_n or W_n (see
- 4951 Fig A.2).
- 4952 – $X0$ – $X30$ are aliases for the whole 64-bit bitvector stored in $R0$ – $R30$.
- 4953 – $W0$ – $W30$ are aliases for the least-significant 32-bit vector stored in $R0$ – $R30$.
- 4954 ▷ a stack pointer, SP .
- 4955 – WSP is an alias for the least-significant 32-bit vector of SP .
- 4956 ▷ a program counter register, named PC , not directly accessible by software.
- 4957 ▷ a collection of ‘special-purpose registers’ which generally store some processor state, e.g.
- 4958 – $NZCV$, the flag register.
- 4959 – $DAIF$, interrupt mask register.
- 4960 – $CurrentEL$, the current exception level register.
- 4961 – ...
- 4962 ▷ a collection of ‘system registers’ which are generally configuration and identification registers, which
- 4963 control how the machine executes, e.g.
- 4964 – $SCTLR_{EL1}$ the system configuration register, for $EL1$ and below.
- 4965 – CTR_{EL0} the cache-type identification register, accessible from $EL0$.
- 4966 – ...

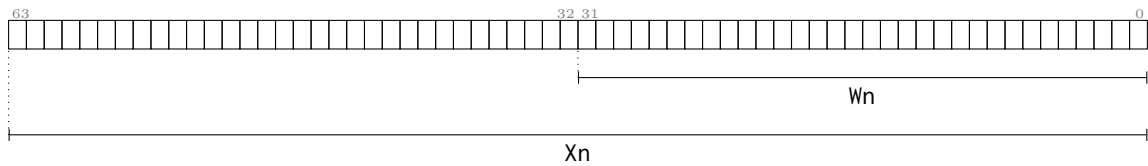


Figure A.2: Views of general-purpose register R_n .

A.2 Guide to Instructions

Chapter contents

A.1	Architectural concepts	203
A.2	Guide to Instructions	205
A.2.1	Branches	205
	B	205
	B.cond	206
	BL and RET	206
	BR and BLR	207
	CBZ and CBNZ	207
A.2.2	Comparisons	208
	CMP	208
A.2.3	Register moving and arithmetic	208
	MOV	208
	MRS and MSR	209
	ADD	209
	EOR	209
	LSL and LSR	209
A.2.4	Memory accesses	210
	LDR	210
	STR	210
	LDP and STP	211
A.2.5	Barriers	211
	DMB	211
	DSB	211
	ISB	212
A.2.6	Cache maintenance	212
	DC	212
	IC	212
A.2.7	TLB maintenance	212
	TLBI-by-address	212
	TLBI-by-ASID	213
	TLBI-ALL	213
A.2.8	Exceptions	213
	SVC and ERET	213

A.2.1 Branches

Branches are those instructions which write to the PC register.

B

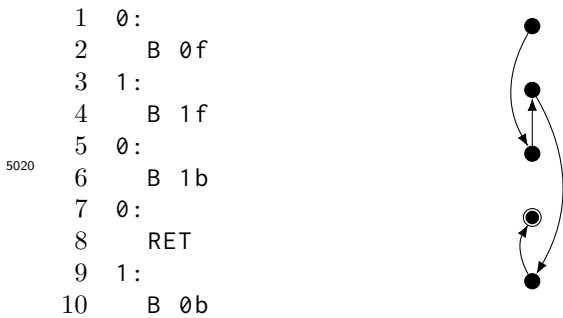
B <LABEL>

Jumps to a given label.

Example The following code branches to label L2, then writes 2 to general-purpose register R0:

```
1  b L2
2  L1:
3    MOV X0, #1
4    RET
5  L2:
6    MOV X0, #2
7    RET
```

5018 **Example** Labels can be numeric, and branches to them can be suffixed with f ('forward') or b ('back').
 5019 e.g. this code has the control-flow-graph shown on the right:



5021 **B.cond**

5022 `B.<COND> <LABEL>`

5023 Jumps to the given label, if the condition given is true.

5024 **Conditions** The conditions are based on the current value of the condition register, NZCV, which are set
 5025 by condition instructions (e.g. CMP), and then the <COND> is one of:

- 5026 ▷ eq: Z==1
- 5027 ▷ ne: Z==0
- 5028 ▷ gt: N==V && Z==0
- 5029 ▷ lt: N!=V

5030 **Example** This program returns 0, as the values are unequal.

```

5031 1 MOV X0,#13
5032 2 MOV X1,#11
5033 3 CMP X0,X1
5034 4 // now NCZV=={0,0,1,1}
5035 5 B.eq Lequal
5036 6 Ldifferent:
5037 7 MOV X0,#0
5038 8 RET
5039 9 Lequal:
5040 10 MOV X0,#1
5041 11 RET
  
```

5042 **BL and RET**

5043 `BL <LABEL>`
 5044 `RET`

5044 Branch-and-link (aka 'call') and return. BL jumps to given label, saving the return location (current
 5045 value of PC + 4), to register X30. RET then returns, by branching to the location stored in general-purpose
 5046 register X30.

5047 **Example** The following example returns 1,2,3 to registers R0, R1 and R2, respectively. Note that the
 5048 general-purpose register X30 is overwritten by BL, so the following example explicitly saves and restores
 5049 the value to some arbitrarily-picked general-purpose registers.

```

5050 1 MOV X20,X30
5051 2 BL f
5052 3 MOV X30,X20
5053 4 RET
  
```

```

5054 5 f:
5055 6   MOV X0,#1
5056 7   MOV X21,X30
5057 8   BL g
5058 9   MOV X30,X21
5059 10  RET
5060 11 g:
5061 12  MOV X1,#2
5062 13  MOV X22,X30
5063 14  BL h
5064 15  MOV X30,X22
5065 16  RET
5066 17 h:
5067 18  MOV X2,#3
5068 19  RET

```

BR and BLR

```

5070                                     BR <GPR>

```

5071 Branches to an address in the given general-purpose register. The address is absolute (not PC-relative).

```

5072                                     BLR <GPR>

```

5073 Branch-and-link register, behaves as BL as before, but jumps to the address stored in the register rather
5074 than to a label.

5075 **Example** The following code places the address of label L into the register R0 using the ADR instruction,
5076 then branches to the label using the stored address:

```

5077 1  ADR X0,L
5078 2  BR L
5079 3  L:
5080 4   MOV X0,#1
5081 5   RET

```

CBZ and CBNZ

```

5083                                     CBZ <GPR>, <LABEL>
5084                                     CBNZ <GPR>, <LABEL>

```

5084 Jumps to given label, if the value in the given general-purpose register is zero (or *not* zero if CBNZ).

5085 **Example** The following code returns with 3 in R3, since X0 is zero (so the first CBNZ) is not taken, X1 is
5086 not zero (so the first CBZ) is not taken, but X2 is zero so the final CBZ is taken, and label L2 is branched to,
5087 and the fallthrough case is missed:

```

5088 1  MOV X0,#0
5089 2  MOV X1,#1
5090 3  MOV X2,#0
5091 4
5092 5  CBNZ X0,L0
5093 6  CBZ X1,L1
5094 7  CBZ X2,L2
5095 8
5096 9  // (fallthrough case)
5097 10 MOV X3,#0
5098 11 RET
5099 12
5100 13 L0:
5101 14  MOV X3,#1
5102 15  RET

```

```

5103 16 L1:
5104 17     MOV X3,#2
5105 18     RET
5106 19 L2:
5107 20     MOV X3,#3
5108 21     RET

```

5109 A.2.2 Comparisons

5110 For use with the B.cond instruction. These instructions write to the NZCV flag register.

5111 CMP

```

5112                                     CMP <GPR0>, <GPR1>
5113                                     CMP <GPR0>, #<IMM>

```

5113 Subtracts the value stored in the second argument (either from a general purpose register or an immediate
5114 value) from the value stored in the first general purpose register, setting the flag register.

5115 **Example** At the end of this program the flag registers are set such that $NCZV==\{0,0,1,0\}$ i.e. the result
5116 is not-negative, no carry, it is zero, and no overflow.

```

5117 1  MOV X0,#100
5118 2  CMP X0,X0

```

5119 **Example** At the end of this program the flag registers are set such that $NCZV==\{1,0,0,0\}$ i.e. the result
5120 is negative, no carry, not zero, and no overflow.

```

5121 1  MOV X0,#1
5122 2  MOV X1,#2
5123 3  CMP X0,X1

```

5124 **Example** At the end of this program the flag registers are set such that $NCZV==\{1,0,0,1\}$ i.e. the result
5125 is negative, no carry, not zero, and it overflowed.

```

5126 1  MOV X0,#0
5127 2  NEG X0,X0
5128 3  CMP X0,#1

```

5129 A.2.3 Register moving and arithmetic

5130 MOV

```

5131                                     MOV <GPR0>, <GPR1>
5132                                     MOV <GPR0>, #<IMM>
5133                                     MOV <GPR0>, #<IMM>, LSL #<IMM>

```

5132 Copies a value stored in the second argument (either in a general-purpose register or a 16-bit immediate
5133 value) into the first argument.

5134 Optionally, the immediate value can be shifted left a multiple of 16.

5135 **Example** At the end of this program X0 contains the value 2, and X1 contains the value 1.

```

5136 1  MOV X0,#1
5137 2  MOV X1,#2
5138 3  MOV X2,#3
5139 4  MOV X2,X0
5140 5  MOV X0,X1
5141 6  MOV X1,X2

```


MRS and MSR

```
MSR <SYSREG>, <GPR>
MRS <GPR>, <SYSREG>
```

Writes (MSR) or reads (MRS) a system (or special-purpose) register.

Example This program sets bits 31-28 to one in the flags register then reads the CTR_EL0 identification register into general-purpose register R1 (note the flags are not relevant here, this is just an example register):

```
1 MOV X0, #0xf000 LSL 16
2 MSR NCZV, X0
3 MRS X1, CTR_EL0
```

ADD

```
ADD <GPR0>, <GPR1>, <GPR2>
ADD <GPR0>, <GPR1>, #<IMM>
```

Adds the values stored in the second and third arguments together, and stores the result in the register given as the first argument.

Examples It is common to pass the same register as input and output to do an increment:

```
1 MOV X0, #1
2 ADD X0, X0, #1
3 // {R0==2}
```

Simple addition:

```
1 MOV X0, #1
2 MOV X1, #2
3 ADD X2, X0, X1
4 // {R2==3}
```

EOR

```
EOR <GPR0>, <GPR1>, <GPR2>
EOR <GPR0>, <GPR1>, #<IMM>
```

Exclusive-or. Does a bitwise exclusive or on the values stored in the second and third arguments, and writes the result to the register given as the first argument.

Examples EOR behaves as a bitwise XOR over integers:

```
1 MOV X0, #3
2 MOV X1, #5
3 EOR X2, X0, X1
4 // {X2==6}
```

Exclusive-or'ing a register with itself zeroes it:

```
1 MOV X0, #13
2 EOR X0, X0, X0
3 // {X0==0}
```

LSL and LSR

```
LSL <GPR0>, <GPR1>, <GPR2>
LSL <GPR0>, <GPR1>, #<IMM>
```

```
LSR <GPR0>, <GPR1>, <GPR2>
LSR <GPR0>, <GPR1>, #<IMM>
```

5179 Logical shift left/right. Shifts the value in the second argument by the amount in the third argument,
5180 and stores the result in the general-purpose register named as the first argument.

5181 **Examples** Left-shifts are multiplication by 2:

```
5182 1 MOV X0,#1  
5183 2 LSL X1,X0,12  
5184 3 // {X1==4096}
```

5185 Right shifts are floor division by 2:

```
5186 1 MOV X0,#5  
5187 2 LSR X1,X0,1  
5188 3 // {X1==2}
```

5189 **A.2.4** Memory accesses

5190 **LDR**

```
LDR <GPR0>, [<GPR1>]  
LDR <GPR0>, [<GPR1>, #<IMM>]
```

5191

```
LDRB <Wn>, [<GPR1>]  
LDRB <Wn>, [<GPR1>, #<IMM>]
```

5192 Reads the value at the memory address stored in the register <GPR1>, and stores the value in register
5193 <GPR0>.

5194 Optionally, an offset to the address can be provided as an immediate value.

5195 There are also address register writeback versions of these instructions, see the full manual.

5196 **NOTE:** if the first argument is a 32-bit alias Wn then a 32-bit value is read from memory, if the first
5197 argument is a 64-bit alias Xn then a 64-bit value is read from memory. If the mnemonic is LDRB then it
5198 loads a single byte, and the register must be a 32-bit alias.

5199 **STR**

```
STR <GPR0>, [<GPR1>]  
STR <GPR0>, [<GPR1>, #<IMM>]
```

5200

```
STRB <Wn>, [<GPR1>]  
STRB <Wn>, [<GPR1>, #<IMM>]
```

5201 Writes the value stored in register named by the <GPR0> argument, into the memory address stored in the
5202 register <GPR1>.

5203 Optionally, an offset to that address can be provided as an immediate value.

5204 **NOTE:** if the first argument is a 32-bit alias Wn then a 32-bit value is written to memory, if the first
5205 argument is a 64-bit alias Xn then a 64-bit value is written to memory.

5206 **Example** The following code writes the 1 as a 64-bit vector to address 1000, the value 2 as a 32-bit
5207 vector to address 1004, and the values 3,4,5 and 6 to addresses 1008,1009,1010, and 1011.

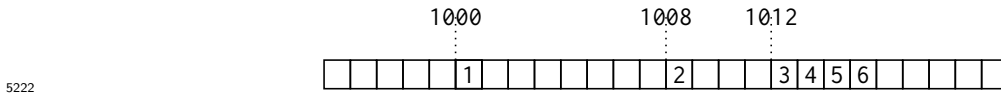
```
5208 1 MOV X0,#1000  
5209 2 MOV X1,#1  
5210 3 MOV W2,#2  
5211 4 MOV W3,#3  
5212 5 MOV W4,#4  
5213 6 MOV W5,#5  
5214 7 MOV W6,#6  
5215 8 STR X1,[X0]
```

```

5216 9  STR W2,[X0,#8]
5217 10 STRB W3,[X0,#12]
5218 11 STRB W4,[X0,#13]
5219 12 STRB W5,[X0,#14]
5220 13 STRB W6,[X0,#15]

```

5221 Resulting in memory like (noting Arm is little-endian by default):



5223 LDP and STP

```

5224 LDP <GPR0>, <GPR1>, [<GPR2>]
5225 STP <GPR0>, <GPR1>, [<GPR2>]

```

5225 Load and store pair variants of the load and store instructions. These read or write from two adjacent 32-
5226 or 64-bit locations, starting at the address stored in <GPR2>, using two separate general-purpose registers
5227 for the data.

5228 A.2.5 Barriers

5229 DMB

```
5230 DMB.<KIND>
```

5231 Data memory barrier.

5232 Arm categorise the kinds into two partitions:

- 5233 ▷ The *types*: whether this orders reads or writes or both.
- 5234 ▷ The *domain*: whether the effect is visible to just this core, or all.

5235 A sample of the kinds used in litmus tests are given below:

Kind	Types	Domain
SY	RW.RW	Full system
ISH	RW.RW	Full system
ST	W.W	Full system
LD	R.RW	Full system

5237 See the full Arm architecture reference manual for the rest.

5238 DSB

```
5239 DSB.<KIND>
```

5240 Data synchronisation barrier. Like a DMB, but affects (some) implicit memory effects, too.

5241 Arm categorise the kinds two ways:

- 5242 ▷ The *types*: whether this orders reads or writes or both.
- 5243 ▷ The *domain*: whether the effect is visible to just this core, or all.

5244 A sample of the kinds used in litmus tests are given below:

Kind	Types	Domain
SY	RW.RW	Full system
ISH	RW.RW	Full system
ST	W.W	Full system
LD	R.RW	Full system
NSH	RW.RW	This CPU only

5246 See the full Arm architecture reference manual for the rest.

5247 ISB

5248 `ISB`

5249 Instruction synchronisation barrier.

5250 A.2.6 Cache maintenance

5251 DC

5252 `DC <OP>, <GPR>`

5253 Data Cache maintenance by address. Performs the cache maintenance operation *OP* to the address stored
5254 in the given general-purpose register.

5255 A sample of the kinds used in litmus tests are given below:

	Kind	Clean/Invalidate	To
5256	CVAU	Clean	Point of Unification
	CVAC	Clean	Point of Coherency
	CIVAC	Clean&Invalidate	Point of Coherency

5257 See the full Arm architecture reference manual for the rest.

5258 IC

5259 `IC <OP>, <GPR>`

5260 Instruction Cache maintenance by address. Performs the cache maintenance operation *OP* to the address
5261 stored in the given general-purpose register.

5262 A sample of the kinds used in litmus tests are given below:

	Kind	Clean/Invalidate	To
5263	IVAU	Invalidate	Point of Unification
	IVAC	Invalidate	Point of Coherency

5264 See the full Arm architecture reference manual for the rest.

5265 `IC <ALLOP>`

5266 Instruction Cache maintenance, not by address.

5267 ALLOP can be one of:

	Op	Clean/Invalidate	To	Domain
5268	IALLU	Invalidate	Point of Unification	This CPU only
	IALLUIS	Invalidate	Point of Unification	All CPUs

5269 A.2.7 TLB maintenance

5270 TLBI-by-address

5271 `TLBI <OP>, <GPR>`

5272 TLB maintenance, by page number stored in the general-purpose register given as argument:



5274 Encoding of TLBI-by-Address argument register.

5275 A sample of TLBI-by-Address operations:

- 5276 ▷ VAE1: by virtual address and ASID, for the EL1&0 regime, for this PE.
- 5277 ▷ VAE1IS: by virtual address and ASID, for the EL1&0 regime, for all PEs.
- 5278 ▷ VAAE1: by virtual address, for all ASIDs, for the EL1&0 regime, for this PE.
- 5279 ▷ VAAE1IS: by virtual address, for all ASIDs, for the EL1&0 regime, for all PEs.

- 5280 ▷ VAE2: by virtual address and ASID, for the EL2 regime, for this PE.
- 5281 ▷ IPAS2E1: by intermediate physical address, for the current VMID, for the EL1&0 regime, for this
- 5282 PE, second stage only.
- 5283 ▷ IPAS2E1IS: by intermediate physical address, for the current VMID, for the EL1&0 regime, for all
- 5284 PEs, second stage only.

5285 TLBI-by-ASID

5286 TLBI ASIDE1, <GPR>
 5287 TLBI ASIDE2, <GPR>

5287 TLB maintenance, for an ASID stored in the general-purpose register given as argument:



5289 Encoding of TLBI-by-ASID argument register.

5290 TLBI-ALL

5291 TLBI <OP>

5292 TLB maintenance, for an ASID stored in the general-purpose register given as argument:

5293 A sample of TLBI-ALL operations:

- 5294 ▷ ALLE1: for any ASID, any VMID, stage 1 and stage2, for the EL1&0 regime, this PE only.
- 5295 ▷ ALLE1IS: for any ASID, any VMID, stage 1 and stage2, for the EL1&0 regime, for all PEs.
- 5296 ▷ ALLE2: for any ASID, for the EL2 regime, for this PE.
- 5297 ▷ VMALLE1IS: for any ASID, for the current VMID, for stage1, for the EL1&0 regime, for all PEs.
- 5298 ▷ VMALLS12E1IS: for any ASID, for the current VMID, for stage1 and stage2, for the EL1&0 regime,
- 5299 for all PEs.

5300 A.2.8 Exceptions

5301 SVC and ERET

5302 SVC #<IMM>

5303 Take an exception right here. Saves the current processor state (current exception level, flags, etc. but not
 5304 register values) to the saved processor status register (SPSR_ELn) and then jumps to the address stored in
 5305 the vector base address register (VBAR_ELn). Jumps to the address stored in the exception link register
 5306 (ELR_ELn), and restores the processor status which was saved on taking the exception (in the SPSR_ELn).

5307 The immediate value is stored in the exception syndrome register, which software can read.

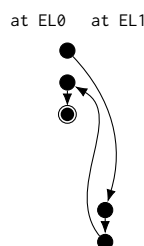
5308 ERET

5309 Return from exception. Jumps to the address stored in the exception link register (ELR_ELn), and restores
 5310 the processor status which was saved on taking the exception (in the SPSR_ELn).

5311 **Example** Execution jumps between process and the exception handler, as shown by the control-flow-graph
 5312 on the right, with columns showing the current exception level.

```

5313 1  MOV X0, #1
2  SVC #0
3  MOV X2, #3
4  RET
5
6  at_VBAR:
7  MOV X1, #2
8  ERET
  
```



The (i)Flat model

This Appendix is based on: *ARMv8-A system semantics: instruction fetch in relaxed architectures [35]* by Ben Simner, Shaked Flur, Christopher Pulte, Alasdair Armstrong, Jean Pichon-Pharabod, Luc Maranget, and Peter Sewell. Published in the proceedings of the 29th European Symposium on Programming (ESOP, 2020).

This appendix reproduces, in full, the Flat model with extensions for instruction fetching (see Chapter 4).

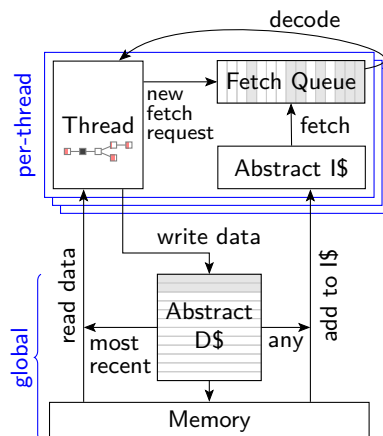
Since the instruction-fetching parts are additive, it can also serve as a reference for the original Flat model of [7], by simply ignoring the ifetch-specific parts.

To help reading this document we have colour-coded some text as follows:

▷ [ifetch] Instruction fetch and cache maintenance instructions

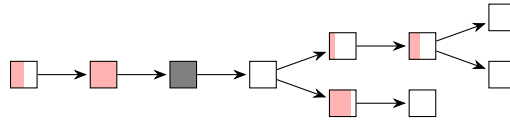
The operational model is expressed as a state machine, with states that are an abstract representation of hardware machine states. We first introduce the model states and transitions informally.

Model states A model state consists just of a shared memory and a tuple of thread model states:



The shared memory state effectively just records the most recent write to each location, together with some additional data for exclusives. To handle instruction fetching, the shared memory is extended with a data cache buffer of all the writes still visible to instruction fetches. Each thread is extended with an instruction cache that can be fetched from and fetch queue of buffered pre-fetched instructions.

Each thread model state consists principally of a list or tree of instruction instances, some of which have been finished, and some of which have not. Below we show an example for a thread that is executing 10 instruction instances. Some (grey) are finished; others (pink) have run some but perhaps not all of their instruction semantics; instructions are not necessarily atomic. Those with multiple children are branch instructions with multiple potential speculative successors being explored simultaneously.



5338

5339 Non-finished instruction instances can be subject to restart, e.g. if they depend on an out-of-order or
 5340 speculative read that turns out to be unsound. The finished instances are not necessarily contiguous: in
 5341 the example, the third is finished even though its predecessors are not, which can only happen if they are
 5342 sufficiently independent. Instruction instances with multiple children are conditional branches for which
 5343 the thread has fetched multiple successors. When a conditional branch is finished, any un-taken
 5344 alternative paths are discarded, and instruction instances that follow (in program order) a non-finished
 5345 conditional branch cannot be finished until that conditional branch is. One can choose whether or not to
 5346 allow simultaneous exploration of multiple successors of a conditional branch (as shown above); this does
 5347 not affect the set of allowed outcomes.

5348 The intra-instruction behaviour of a single instruction can largely be treated as sequential (but not atomic)
 5349 execution of its ASL/Sail pseudocode. Each instruction instance state includes a pseudocode execution
 5350 state, which one can think of as a representation of the pseudocode control state, pseudocode call stack,
 5351 and local variable values. An instruction instance state also includes information, detailed below, about
 5352 the instruction instance's memory and register footprints, its register and memory reads and writes,
 5353 whether it is finished, etc.

5354 **Model transitions** For any state, the model defines the set of allowed transitions, each of which is a
 5355 single atomic step to a new abstract machine state. Each transition arises from the next step of a single
 5356 instruction instance; it will change the state of that instance, and it may depend on or change the rest
 5357 of its thread state and/or the shared memory state. Instructions cannot be treated as atomic units:
 5358 complete execution of a single instruction instance may involve many transitions, which can be interleaved
 5359 with those of other instances in the same or other threads, and some of this is programmer-visible. The
 5360 transitions are introduced below and defined in §B.0.4, with a precondition and a construction of the
 5361 post-transition model state for each. The transitions labelled \circ can always be taken eagerly, as soon as
 5362 they are enabled, without excluding other behaviour; the \triangleright cannot.

5363 Transitions for all instructions:

- 5364 \circ [\[ifetch \] Fetch request](#): This transition speculates the next address as a po-successor of a previously
 5365 speculated instruction.
- 5366 \triangleright [Fetch instruction](#): Satisfy the fetch from instruction memory.
- 5367 \circ [\[ifetch \] Decode instruction](#): [Decode the instruction](#).
- 5368 \circ [Register read](#): This is a read of a register value from the most recent program-order predecessor
 5369 instruction instance that writes to that register.
- 5370 \circ [Register write](#)
- 5371 \circ [Pseudocode internal step](#): this covers ASL/Sail internal computation, function calls, etc.
- 5372 \circ [Finish instruction](#): At this point the instruction pseudocode is done, the instruction cannot be
 5373 restarted or discarded, and all memory effects have taken place. For a conditional branch, any
 5374 non-taken po-successor branches are discarded.

5375 Load instructions:

- 5376 \circ [Initiate memory reads of load instruction](#): At this point the memory footprint of the load is
 5377 provisionally known and its individual reads can start being satisfied.
- 5378 \triangleright [Satisfy memory read by forwarding from writes](#): This partially or entirely satisfies a single read by
 5379 forwarding from its po-previous writes.
- 5380 \triangleright [Satisfy memory read from memory](#): This entirely satisfies the outstanding slices of a single read,
 5381 from memory.
- 5382 \circ [Complete load instruction \(when all its reads are entirely satisfied\)](#): At this point all the reads of
 5383 the load have been entirely satisfied and the instruction pseudocode can continue execution. A load
 5384 instruction can be subject to being restarted until the [Finish instruction](#) transition. In some cases it
 5385 is possible to tell that a load instruction will not be restarted or discarded before that, e.g. when all
 5386 the instructions po-before the load instruction are finished. The [Restart condition](#) over-approximates
 5387 the set of instructions that might be restarted.

5388 Store instructions:

- 5389 ○ **Initiate memory writes of store instruction, with their footprints:** At this point the memory footprint
- 5390 of the store is provisionally known.
- 5391 ○ **Instantiate memory write values of store instruction:** At this point the writes have their values and
- 5392 program-order-subsequent reads can be satisfied by forwarding from them.
- 5393 ○ **Commit store instruction:** At this point the store is guaranteed to happen (it cannot be restarted or
- 5394 discarded), and the writes can start being propagated to memory.
- 5395 ▷ **Propagate memory write:** This propagates a single write to memory.
- 5396 ○ **Complete store instruction (when its writes are all propagated):** At this point all writes have been
- 5397 propagated to memory, and the instruction pseudocode can continue execution.

5398 Barrier instructions:

- 5399 ○ **Commit barrier**

5400 Cache maintenance instructions:

- 5401 ▷ [ifetch] **Begin IC: Initiate instruction cache maintenance.**
- 5402 ○ [ifetch] **Propagate IC to thread: Do instruction cache maintenance for a specific thread.**
- 5403 ▷ [ifetch] **Perform DC: Clean the abstract data cache for a specific cache line.**

5404 Instruction cache updates:

- 5405 ○ [ifetch] **Add to instruction cache for thread: Update instruction cache for thread with write.**

5406 **B.0.1 Intra-instruction Pseudocode Execution**

5407 To link the model transitions introduced above to the execution of the instructions an interface is needed
 5408 between Sail and the rest of the concurrency model. For each instruction instance this intra-instruction
 5409 semantics is expressed as a state machine, essentially running the instruction pseudocode, where each
 5410 pseudocode execution state is a request of one of the following forms:

5411	READ_MEM(<i>read_kind</i> , <i>address</i> , <i>size</i> , <i>read_continuation</i>)	Read request
	PERFORM_IC(<i>address</i> , <i>res_continuation</i>)	Propagate an ic ivau
	WAIT_IC(<i>address</i> , <i>res_continuation</i>)	Wait for an ic ivau to complete
	PERFORM_DC(<i>address</i> , <i>res_continuation</i>)	Propagate a dc cvau
	WRITE_EA(<i>write_kind</i> , <i>address</i> , <i>size</i> , <i>next_state</i>)	Write effective address
5412	WRITE_MEMV(<i>memory_value</i> , <i>write_continuation</i>)	Write value
	BARRIER(<i>barrier_kind</i> , <i>next_state</i>)	Barrier
	READ_REG(<i>reg_name</i> , <i>read_continuation</i>)	Register read request
	WRITE_REG(<i>reg_name</i> , <i>register_value</i> , <i>next_state</i>)	Write register
	INTERNAL(<i>next_state</i>)	Pseudocode internal step
	DONE	End of pseudocode

5413 Each of these states is a suspended computation with a request for an action or input from the concurrency
 5414 model and, except in the case of DONE, a continuation for the remaining execution.

5415 Here memory values are lists of bytes, addresses are 64-bit numbers, read and write kinds identify whether
 5416 they are regular, exclusive, and/or release/acquire operations, register names identify a register and
 5417 slice thereof (start and end bit indices), and the continuations describe how the instruction instance will
 5418 continue for any value that might be provided by the surrounding memory model. This largely follows
 5419 [15, §2.2], except that memory writes are split into two steps, WRITE_EA and WRITE_MEMV. We ensure
 5420 these are paired in the pseudocode, but there may be other steps between them: it is observable that
 5421 the WRITE_EA can occur before the value to be written is determined, because the potential memory
 5422 footprint of the instruction becomes provisionally known then.

5423 We ensure that each instruction has at most one memory read, memory write, or barrier step, by rewriting
 5424 the pseudocode to coalesce multiple reads or writes, which are then split apart into the architecturally
 5425 atomic units by the thread semantics; this gives a single commit point for all memory writes of an
 5426 instruction.

5427 Each bit of a register read should be satisfied from a register write by the most recent (in program order)
 5428 instruction instance that can write that bit, or from the thread's initial register state if there is no such.

5429 That instance may not have executed its register write yet, in which case the register read should block.
 5430 The semantics therefore has to know the register write footprint of each instruction instance, which it
 5431 calculates when the instruction instance is created. We ensure in the pseudocode that each instruction
 5432 does exactly one register write to each bit of its register footprint, and also that instructions do not do
 5433 register reads from their own register writes. In some cases, but not in the fragment of ARM that we
 5434 cover at present, register write footprints need to be dynamically recalculated, when the actual footprint
 5435 only becomes known during pseudocode execution.

5436 Data-flow dependencies in the model emerge from the fact that a register read has to wait for the
 5437 appropriate register write to be executed (as described above). This has to be carefully handled in order
 5438 not to create unintentional strength. First, for some instructions we need to ensure that the pseudocode
 5439 is in the maximally liberal order, e.g. to allow early computed-address register writebacks before the
 5440 corresponding memory write. Leaving load-pair aside (which we do not cover), and the treatment of the
 5441 multiple reads or writes that can be associated with a single load or store instruction (which we do), we
 5442 have not so far needed other intra-instruction concurrency. Second, the model has to be able to know
 5443 when a register read value can no longer change (i.e. due to instruction restart). We approximate that
 5444 by recording, for each register write, the set of register and memory reads the instruction instance has
 5445 performed at the point of executing the write. This information is then used as follows to determine
 5446 whether a register read value is final: if the instruction instance that performed the register write from
 5447 which the register reads from is finished, the value is final; otherwise check that the recorded reads for the
 5448 register write do not include memory reads, and continue recursively with the recorded register reads. For
 5449 the instructions we cover this approximation is exact.

5450 We express the pseudocode execution semantics in two ways: a definitional interpreter for Sail [15], with
 5451 an exhaustive symbolic mode to (re)calculate an instruction’s memory and register footprints, and as a
 5452 shallow embedding, translating Sail into directly executable code, with separate hand-written definitions
 5453 of the footprint functions. The two are essentially equivalent: the first lets one small-step through the
 5454 pseudocode interactively, while the second is more efficient and should be more convenient for proof.

5455 **B.0.2 Instruction Instance States**

5456 Each instruction instance i has a state comprising:

- 5457 ▷ *program_loc*, the memory address from which the instruction was fetched;
- 5458 ▷ *instruction_kind*, identifying whether this is a load, store, or barrier instruction, each with the
 5459 associated kind; or a conditional branch; or a ‘simple’ instruction.
- 5460 ▷ *regs_in*, the set of input *reg_names*, as statically determined;
- 5461 ▷ *regs_out*, the output *reg_names*, as statically determined;
- 5462 ▷ *pseudocode_state* (or sometimes just ‘state’ for short), one of
 - 5463 – *PLAIN_next_state*, ready to make a pseudocode transition;
 - 5464 – *PENDING_MEM_READS* k , performing the read(s) from memory of a load; or
 - 5465 – *PENDING_MEM_WRITES* k , performing the write(s) to memory of a store;
- 5466 ▷ *reg_reads*, the accumulated register reads, including their sources and values, of this instance’s
 5467 execution so far;
- 5468 ▷ *reg_writes*, the accumulated register writes, including dependency information to identify the register
 5469 reads and memory reads (by this instruction) that might have affected each;
- 5470 ▷ *mem_reads*, a set of memory read requests. Each request includes a memory footprint (an address
 5471 and size) and, if the request has already been satisfied, the set of write slices (each consisting of a
 5472 write and a set of its byte indices) that satisfied it.
- 5473 ▷ *mem_writes*, a set of memory write requests. Each request includes a memory footprint and, when
 5474 available, the memory value to be written. In addition, each write has a flag that indicates whether
 5475 the write has been propagated (passed to the memory) or not.
- 5476 ▷ information recording whether the instance is committed, finished, etc.

5477 Read requests include their read kind and their memory footprint (their address and size), the as-yet-
 5478 unsatisfied slices (the byte indices that have not been satisfied), and, for the satisfied slices, information
 5479 about the write(s) that they were satisfied from. Write requests include their write kind, their memory
 5480 footprint, and their value. When we refer to a write or read request without mentioning the kind of
 5481 request we mean the request can be of any kind. A load instruction which has initiated (so its read request

list *mem_reads* is not empty) and for which all its read requests are satisfied (i.e. there are no unsatisfied slices) is said to be *entirely satisfied*.

5484 **B.0.3 Thread States**

5485 The model state of a single hardware thread includes:

- 5486 ▷ *thread_id*, a unique identifier of the thread;
- 5487 ▷ *register_data*, the name, bit width, and start bit index for each register;
- 5488 ▷ *initial_register_state*, the initial register value for each register;
- 5489 ▷ *initial_fetch_address*, the initial fetch address for this thread;
- 5490 ▷ *instruction_tree*, a tree or list of the instruction instances that have been fetched (and not discarded),
- 5491 in program order.

5492 **B.0.4 Model Transitions**

5493 **Fetch request** For some instruction *i*, any possible next fetch address *loc* can be requested, adding it to
5494 the fetch queue, if:

- 5495 1. it has not already been requested, i.e., none of the immediate successors of *i* in the thread's
5496 *instruction_tree* are from *loc*; and
- 5497 2. either *i* is not decoded, or, if it has been, *loc* is a possible next fetch address for *i*:
 - 5499 (a) for a non-branch/jump instruction, the successor instruction address (*i.program_loc+4*);
 - 5500 (b) for a conditional branch, either the successor address or the branch target address¹; or
 - 5501 (c) for a jump to an address which is not yet determined, any address (this is approximated in our
5502 tool implementation, necessarily).

5503 Action: add an unfetched entry for *loc* to the fetch queue for *i*'s thread.

5504 Note that this allows speculation past conditional branches and calculated jumps.

5505 **Fetch instruction (ifetch)** In *ifetch* mode this transition replaces the original 'Fetch instruction' transi-
5506 tion.

5507 For any fetch-queue entry in the UNFETCHED state, its fetch can be satisfied from the instruction cache,
5508 from write-slices *ws*, if:

- 5509 1. the write-slices (parts of writes) *ws* have the 4-byte footprint of the entry and can be constructed
5510 from a write in the instruction cache.

5512 Action: change the fetch-queue entry's state to FETCHED(*ws*).

5513 **Fetch instruction (unpredictable)** For any fetch-queue entry in the UNFETCHED state, its fetch can be
5514 satisfied from the instruction cache in a constrained-unpredictable way, if:

- 5515 1. there exists a set of write-slices, each of which can be constructed in the same way as above;
- 5517 2. that set contains write-slices corresponding to distinct opcodes, and at least one of those is an
5518 instruction that is not B.cond or one of {B, BL, BRK, HVC, SMC, SVC, ISB, NOP}, and they are not all
5519 B.cond instructions.

5520 Action: record the fetch-queue entry as CONSTRAINED_UNPREDICTABLE. When this has reached decode
5521 and the corresponding point in the instruction tree becomes non-speculative, the entire thread state will
5522 become CONSTRAINED_UNPREDICTABLE.

5523 **Fetch instruction (B.cond)** For any fetch-queue entry in the UNFETCHED state, its fetch can be satisfied
5524 from the instruction cache, from write-slices *ws* and *ws'*, with value *ws''*, if:

- 5525 1. there exists write-slices *ws* and *ws'*, each of which can be constructed in the same way as above;
- 5527 2. *ws* and *ws'* correspond to the encoding of two conditional branch instructions *b* and *b'*;
- 5528 3. the write-slices *ws''* can be constructed as the combination of *ws* and *ws'* such that *ws''* is the
5529 encoding of the branch instruction with *b*'s condition and *b'*'s target.

¹In AArch64, all the conditional branch instructions have statically determined addresses.

5530 Action: record the fetch-queue entry as `FETCHED(ws)`.

5531 **Decode instruction** If the last entry in the fetch queue is in `FETCHED(ws)` state, it can be removed from
5532 the queue, decoded, and begin execution, if all po-previous ISB instructions in the instruction tree have
5533 finished.

5534 Action:

5535 1. Construct a new instruction instance *i* with the correct instruction kind and state, for *i*'s program
5536 location, and add it to the instruction tree.

5537 2. Discard all speculative entries in the instruction tree that are successors of *i* that are now known to
5538 be incorrect speculations.

5539 Note that this transition is a proxy for the point the instructions will be decoded, but that it is the
5540 intra-instruction semantics that actually performs the decoding, with this transition merely starting the
5541 execution of the pseudocode.

5542 **Fetch instruction** In ifetch mode this transition is replaced by Fetch instruction (ifetch).

5543 A possible program-order successor of instruction instance *i* can be fetched from address *loc* if:

5544 1. it has not already been fetched, i.e., none of the immediate successors of *i* in the thread's *instruc-*
5545 *tion_tree* are from *loc*;

5546 2. *loc* is a possible next fetch address for *i*:

5547 (a) for a non-branch/jump instruction, the successor instruction address (*i.program_loc+4*);

5548 (b) for an instruction that has performed a write to the program counter register (`_PC`), the value
5549 that was written;

5550 (c) for a conditional branch, either the successor address or the branch target address¹; or

5551 (d) for a jump to an address which is not yet determined, any address (this is approximated in our
5552 tool implementation, necessarily); and

5553 3. there is a decodable instruction in program memory at *loc*.

5554 Note that this allows speculation past conditional branches and calculated jumps.

5555 Action: construct a freshly initialized instruction instance *i'* for the instruction in the program memory at
5556 *loc*, including the static information available from the ISA model such as its *instruction_kind*, *regs_in*,
5557 and *regs_out*, and add *i'* to the thread's *instruction_tree* as a successor of *i*.

5558 This involves only the thread, not the storage subsystem, as we assume a fixed program rather than
5559 modelling fetches with memory reads; we do not model self-modifying code.

5560 **Initiate memory reads of load instruction** An instruction instance *i* with next state
5561 `READ_MEM(read_kind, address, size, k)` can initiate the corresponding memory reads. Action:

5562 1. Construct the appropriate read requests *rrs*:

5563 ▷ if *address* is aligned to *size* then *rrs* is a single read request of *size* bytes from *address*;

5564 ▷ otherwise, *rrs* is a set of *size* read requests, each of one byte, from the addresses *ad-*
5565 *dress...address+size-1*.

5566 2. set *i.mem_reads* to *rrs*; and

5567 3. update the state of *i* to `PENDING_MEM_READS k`.

5568 **Complete load instruction (when all its reads are entirely satisfied)** A load instruction instance *i*
5569 in state `PENDING_MEM_READS k` can be completed (not to be confused with finished) if all the read
5570 requests *i.mem_reads* are entirely satisfied (i.e., there are no unsatisfied slices).

5571 Action: update the state of *i* to `PLAIN (read_cont (memory_value))`, where *memory_value* is assembled
5572 from all the write slices that satisfied *i.mem_reads*.

¹In AArch64, all the conditional branch instructions have statically determined addresses.

5575 **Initiate memory writes of store instruction, with their footprints** An instruction instance i with next
5576 state $WRITE_EA(write_kind, address, size, next_state')$ can announce its pending write footprint. Action:

- 5577 1. construct the appropriate write requests:
5578 ▷ if $address$ is aligned to $size$ then ws is a single write request of $size$ bytes to $address$;
5579 ▷ otherwise ws is a set of $size$ write requests, each of one byte size, to the addresses $ad-$
5580 $dress \dots address + size - 1$.
5581 2. set $i.mem_writes$ to ws ; and
5582 3. update the state of i to $PLAIN\ next_state'$.
5583

5584 Note that at this point the write requests do not yet have their values. This state allows non-overlapping
5585 po-following writes to propagate.

5586 **Instantiate memory write values of store instruction** An instruction instance i with next state
5587 $WRITE_MEMV(memory_value, k)$ can initiate the corresponding memory writes. Action:

- 5588 1. split $memory_value$ between the write requests $i.mem_writes$; and
5589 2. update the state of i to $PENDING_MEM_WRITES\ k$.
5590

5591 **Commit barrier** A barrier instruction i in state $PLAIN(next_state)$ where $next_state$ is
5592 $BARRIER(barrier_kind, next_state')$ can be committed if:

- 5593 1. all po-previous conditional branch instructions are finished;
5594 2. all po-previous `dmb sy` barriers are finished;
5595 3. `[ifetch]` all po-previous `dsb sy` barriers are finished; and
5596 4. if i is an `isb` instruction, all po-previous memory access instructions have fully determined memory
5597 footprints; and
5598 5. if i is a `dmb sy` instruction, all po-previous memory access instructions and barriers are finished;;
5599 and
5600 6. `[ifetch]` if i is a `dsb sy` instruction, all po-previous memory access instructions, barriers, and cache
5601 maintenance instructions have finished.
5602

5603 Action:

- 5604 1. Update the state of i to $PLAIN(next_state')$;
5605 2. `[ifetch]` If i is an `isb` instruction, for any instruction instance in this thread's instruction tree, if that
5606 instruction instance is in the `FETCHED` state, set it to the `UNFETCHED` state.
5607

5608 Note that this corresponds to an `ISB` discarding any already-fetched entries from the fetch queue.

5609 **Satisfy memory read by forwarding from writes** For a load instruction instance i in state $PEND-$
5610 $ING_MEM_READS(k)$, and a read request, r in $i.mem_reads$ that has unsatisfied slices, the read request
5611 can be partially or entirely satisfied by forwarding from unpropagated writes by store instruction instances
5612 that are po-before i , if the *read-request-condition* predicate holds. This is if:

- 5613 1. `[ifetch]` all po-previous `dsb sy` instructions are finished; and
5614 2. all po-previous `dmb sy` and `isb` instructions are finished.
5615

5616 Let wss be the maximal set of unpropagated write slices from store instruction instances po-before i , that
5617 overlap with the unsatisfied slices of r , and which are not superseded by intervening stores that are either
5618 propagated or read from by this thread. That last condition requires, for each write slice ws in wss from
5619 instruction i' :

- 5620 ▷ that there is no store instruction po-between i and i' with a write overlapping ws , and
5621 ▷ that there is no load instruction po-between i and i' that was satisfied from an overlapping write
5622 slice from a different thread.

5623 Action:

- 5624 1. Update r to indicate that it was satisfied by wss .
5625 2. Restart any speculative instructions which have violated coherence as a result of this, i.e., for every
5626 non-finished instruction i' that is a po-successor of i , and every read request r' of i' that was satisfied
5627 from wss' , if there exists a write slice ws' in wss' , and an overlapping write slice from a different
5628 write in wss , and ws' is not from an instruction that is a po-successor of i , or if i' was a `data-cache`
5629

5630 maintenance by virtual address to a cache line that overlaps with any of the write slices in wss' ,
5631 restart i' and its data-flow dependents.

5632 **Satisfy memory read from memory** For a load instruction instance i in state `PENDING_MEM_READS(k)`,
5633 and a read request r in $i.mem_reads$, that has unsatisfied slices, the read request can be satisfied from
5634 memory, if:

5635 1. the read-request-condition holds (see previous transition).

5636 Action:

5637 let wss be the write slices from memory or the abstract data cache, whichever is newer, covering the
5638 unsatisfied slices of r , and apply the action of **Satisfy memory read by forwarding from writes**.

5640 Note that **Satisfy memory read by forwarding from writes** might leave some slices of the read request
5641 unsatisfied. **Satisfy memory read from memory**, on the other hand, will always satisfy all the unsatisfied
5642 slices of the read request.

5643 **Commit store instruction** For an uncommitted store instruction i in state `PENDING_MEM_WRITES(k)`,
5644 i can commit if:

- 5645 1. i has fully determined data (i.e., the register reads cannot change);
- 5646 2. all po-previous conditional branch instructions are finished;
- 5647 3. all po-previous `dmb sy` and `isb` instructions are finished;
- 5648 4. `[ifetch]` all po-previous `dsb sy` instructions are finished;
- 5649 5. all po-previous store instructions have initiated and so have non-empty mem_writes ;
- 5650 6. all po-previous memory access instructions have a fully determined memory footprint; and
- 5651 7. all po-previous load instructions have initiated and so have non-empty mem_reads .

5652 Action: record i as committed.

5654 **Propagate memory write** For an instruction i in state `PENDING_MEM_WRITES(k)`, and an unpropagated
5655 write, w in $i.mem_writes$, the write can be propagated if:

- 5656 1. all memory writes of po-previous store instructions that overlap w have already propagated;
- 5657 2. all read requests of po-previous load instructions that overlap with w have already been satisfied,
5658 and the load instruction is non-restartable; and
- 5659 3. all read requests satisfied by forwarding w are entirely satisfied.

5660 Action:

- 5662 1. Restart any speculative instructions which have violated coherence as a result of this, i.e., for every
5663 non-finished instruction i' po-after i and every read request r' of i' that was satisfied from wss' , if
5664 there exists a write slice ws' in wss' that overlaps with w and is not from w , and ws' is not from a
5665 po-successor of i , or if i' is a data-cache maintenance instruction to a cache line whose footprint
5666 overlaps with w , restart i' and its data-flow dependents.
- 5667 2. Record w as propagated.
- 5668 3. Add w as a complete slice to the abstract data cache.

5670 **Complete store instruction (when its writes are all propagated)** A store instruction i in state
5671 `PENDING_MEM_WRITES(k)`, for which all the memory writes in $i.mem_writes$ have been propagated, can
5672 be completed.

5673 Action:

5674 Update the state of i to `PLAIN($k(true)$)`.

5676 **Begin IC** An instruction i (with unique instruction instance ID $iiid$) in state `PERFORM_IC($address$,`
5677 `$state_cont$)` can begin performing the IC behaviour if all po-previous DSB ISH instructions have finished.

5678 Action:

- 5679 1. For each thread tid' (including this one), add $(iiid, address)$ to that thread's ic_writes ;
- 5680 2. Set the state of i to `PROPAGATE_IC($address, state_cont$)`.

5682 **Propagate IC to thread** An instruction i (with ID $iiid$) in state $WAIT_IC(address, state_cont)$ can do
 5683 the relevant invalidate for any thread tid' , modifying that thread's instruction cache and fetch queue, if
 5684 there exists a pending entry $(iiid, address)$ in that thread's ic_writes .

5685 Action:

- 5686 1. For any entry in the fetch queue for thread tid , whose $program_loc$ is in the same minimum-size
 5687 instruction cache line as $address$, and is in $FETCHED(_)$ state, set it to the $UNFETCHED$ state.
 5688 2. For the instruction cache of thread tid , remove any write-slices which are in the same instruction
 5689 cache line of minimum size as $address$.
 5690

5691 **Complete IC** An instruction i (with instruction instance ID $iiid$) in the state $WAIT_IC(address,$
 5692 $state_cont)$ can complete if there exists no entry for $iiid$ in any thread's ic_writes .

5693 Action: set the state of i to $PLAIN(state_cont)$.

5694 **Perform DC** An instruction i in the state $PERFORM_DC(address, state_cont)$ can complete if all
 5695 po-previous DMB ISH and DSB ISH instructions have finished.

5696 Action:

- 5697 1. For the most recent write slices wss which are in the same data cache line of minimum size in the
 5698 abstract data cache as $address$, update the memory with wss .
 5699 2. Remove all those writes from the abstract data cache.
 5700 3. Set the state of i to $PLAIN(state_cont)$.
 5701

5702 **Add to instruction cache for thread** A thread tid 's instruction cache can be spontaneously updated
 5703 with a write w from the storage subsystem, if this write (as a single slice) does not already exist in the
 5704 instruction cache.

5705 Action: Add this write (as a single slice) to thread tid 's instruction cache.

5706 **Register read** An instruction instance i with next state $READ_REG(reg_name, k)$ can do a register read
 5707 if every instruction instance that it needs to read from has already performed the expected register write.

5708 Let $read_sources$ include, for each bit of reg_name , the write to that bit by the most recent (in program
 5709 order) instruction instance that can write to that bit, if any. If there is no such instruction, the source
 5710 is the initial register value from $initial_register_state$. Let $register_value$ be the assembled value from
 5711 $read_sources$. Action:

- 5712 1. add reg_name to $i.reg_reads$ with $read_sources$ and $register_value$; and
 5713 2. update the state of i to $PLAIN(k(register_value))$.
 5714

5715 **Register write** An instruction instance i with next state $WRITE_REG(reg_name, register_value,$
 5716 $next_state')$ can do the register write. Action:

- 5717 1. add reg_name to $i.reg_writes$ with $write_deps$ and $register_value$; and
 5718 2. update the state of i to $PLAIN next_state'$.
 5719

5720 where $write_deps$ is the set of all $read_sources$ from $i.reg_reads$ and a flag that is set to true if i is a load
 5721 instruction that has already been entirely satisfied.

5722 **Pseudocode internal step** An instruction instance i with next state $INTERNAL(next_state')$ can do that
 5723 pseudocode-internal step. Action: update the state of i to $PLAIN next_state'$.

5724 **Finish instruction** A non-finished instruction i with next state $DONE$ can be finished if:

- 5725 1. if i is a load instruction:
 5726 (a) all po-previous dmb sy and isb instructions are finished;
 5727 (b) it is guaranteed that the values read by the read requests of i will not cause coherence violations,
 5728 i.e., for any po-previous instruction instance i' , let cfp be the combined footprint of propagated
 5729 writes from store instructions po-between i and i' and fixed writes that were forwarded to i
 5730 from store instructions po-between i and i' including i' , and let cfp' be the complement of cfp
 5731 in the memory footprint of i . If cfp' is not empty:
 5732

- 5733 i. i' has a fully determined memory footprint;
- 5734 ii. i' has no unpropagated memory write that overlaps with cfp' ; and
- 5735 iii. If i' is a load with a memory footprint that overlaps with cfp' , then all the read requests of
- 5736 i' that overlap with cfp' are satisfied and i' can not be restarted (see §B.0.5).

5737 Here a memory write is called fixed if it is the write of a store instruction that has fully
5738 determined data.

- 5739 2. i has fully determined data; and
- 5740 3. all po-previous conditional branches are finished.

5741 Action:

- 5742 1. if i is a branch instruction, discard any untaken path of execution, i.e., remove any (non-finished)
- 5743 instructions that are not reachable by the branch taken in *instruction_tree*; and
- 5744 2. record the instruction as finished, i.e., set *finished* to *true*.
- 5745

5746 B.0.5 Auxiliary Definitions

5747 **Fully determined** An instruction is said to have fully determined footprint if the memory reads feeding
5748 into its footprint are finished: A register write w , of instruction i , with the associated *write_deps* from
5749 *i.reg_writes* is said to be *fully determined* if one of the following conditions hold:

- 5750 1. i is finished; or
- 5751 2. the load flag in *write_deps* is *false* and every register write in *write_deps* is fully determined.

5752 An instruction i is said to have *fully determined data* if all the register writes of *read_sources* in *i.reg_reads*
5753 are fully determined. An instruction i is said to have a *fully determined memory footprint* if all the register
5754 writes of *read_sources* in *i.reg_reads* that are associated with registers that feed into i 's memory access
5755 footprint are fully determined.

5756 **Restart condition** To determine if instruction i might be restarted we use the following recursive
5757 condition: i is a non-finished instruction and at least one of the following holds,

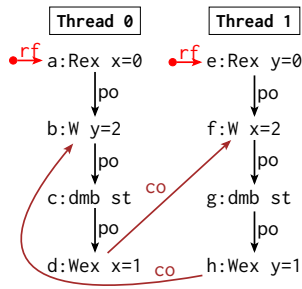
- 5758 1. there exists an unpropagated write w such that applying the action of the [Propagate memory write](#)
- 5759 transition to s will result in the restart of i ;
- 5760 2. there exists a non-finished load instruction l such that applying the action of the [Satisfy memory](#)
- 5761 [read from memory](#) transition to l will result in the restart of i (even if l is already entirely satisfied);
- 5762 or
- 5763 3. there exists a non-finished instruction i' that might be restarted and i is in its data-flow dependents.

5764 **Cache Line of Minimum Size** Cache maintenance operations work over entire cache lines, not individual
5765 addresses. Each address is associated with at least one cache line for the data (and unified) caches, and
5766 one for the instruction caches. The cache line of minimum size is the smallest possible cache line for each
5767 of these. The `CTR_EL0.{DMinLine, IMinLine}` values describe the cache lines of minimum size for the
5768 data and instruction caches as \log_2 of the number of words in the cache line.

5769 B.0.6 Remarks about load/store exclusive instructions

5770 The MCA ARMv8 architecture intends that the success bit of store exclusives does not introduce
5771 dependencies, to allow (e.g.) hardware optimisations that dynamically replace load/store exclusive pairs
5772 by atomic read-modify-write operations that can execute in the memory subsystem and therefore be
5773 guaranteed to succeed. The ARMv8-axiomatic definition assumes all address/data/control dependencies to
5774 be from reads, not writes. In the operational model, matching this weakness has proved to be difficult: it
5775 means the operational model must be able to promise the success or failure of a store-exclusive instruction
5776 even before any of its registers reads/writes have been done, so before the store-exclusive's address and
5777 data are available. The early success promises are the source of deadlocks in the operational model. To
5778 illustrate this consider, for example, the following litmus test and a state where both a and e are satisfied
5779 and finished, and where b and f are not propagated. Then d can promise its success, locking memory
5780 location x, and h can promise its success, locking location y. But now there is a deadlock:

- 5781 ▷ For d to propagate c has to be committed and hence b propagated.
- 5782 But b cannot propagate since y is locked.



5783 ▷ For h to propagate g has to be committed and hence f propagated.
 5784 But f cannot propagate since x is locked.

5785 Similar situations arise from cases where there are other barriers or release/acquire instructions in-between
 5786 the load and the store exclusive, or if the store exclusive has additional dependencies that the load
 5787 exclusive does not have. These are cases that are not really intended to be supported by the architecture.

5788 The model can also currently deadlock if a load and a store-exclusive are paired successfully but later
 5789 turn out to have different addresses: if the store-exclusive promises its success before its address is known
 5790 it locks the matched load-exclusive's memory location; when they later turns out to be to a different
 5791 addresses it never unlocks it. This issue can be fixed, but it is currently still being clarified what exactly
 5792 the architecturally allowed behaviour should be.

Test format: system-litmus-harness

The test format supports writing a variety of kinds of pagetable tests, through both the initial state setup and the data passed from the harness allocator via the `litmus_test_run` data struct.

The data struct contains, for each global variable (e.g. `x`): the virtual address (`%[x]`); the initial last-level descriptor (`%[xdesc]`); the address of the last-level entry (`%[xppte]`); the address of the entry at level `N` (`%[xppteN]`); the page index, e.g. for arguments to TLB maintenance (`%[xpage]`). With some aliases for the different levels to match Linux terminology: `%[xpm2]` for the level 2 entry (`xppte2`); `%[xpud]` for the level 1 entry (`xppte1`).

The initial state enables specifying a rich variety of related machine states, each `INIT_STATE` can include directives for the initial value of the variable:

- ▷ `INIT_UNMAPPED(var)`: that the pagetable entry for `var` starts out invalid.
- ▷ `INIT_VAR(var, value)`: that `var` starts out mapped and the location at its physical address starts out containing `value`.
- ▷ `INIT_ALIAS(var1, var2)`: that `var1` and `var2` should be aliased to the same location.

The programmer can also choose the initial permissions and memory attributes the variables are mapped with:

- ▷ `INIT_PERMISSIONS(var, prot, value)`: that `var` should be mapped with field `prot` set to `value`:
 - for `prot=PROT_AP`, `value` can be any int, but there are some helpful aliases:
 - * `PROT_AP_RWX_X (0x0)`: read-write-execute at EL1, execute only at EL0.
 - * `PROT_AP_RW_RWX (0x1)`: read-write at EL1, read-write-execute at EL0.
 - * `PROT_AP_RX_X (0x2)`: read-execute at EL1, execute only at EL0.
 - * `PROT_AP_RX_RX (0x3)`: read-execute at EL1 and EL0.
 - for `prot=PROT_ATTRIDX`, `value` defines the memory attributes as the index to the default MAIR value, and can be any of:
 - * `PROT_ATTR_DEVICE_nGnRnE (0)`: use strongly-ordered device memory.
 - * `PROT_ATTR_DEVICE_GRE (1)`: standard device memory (with re-ordering, gathering and early write acknowledgement).
 - * `PROT_ATTR_NORMAL_NC (2)`: normal non-cacheable memory.
 - * `PROT_ATTR_NORMAL_RA_WA (3)`: normal cacheable memory.
 - * indexes 4-7 are unused.
- ▷ `INIT_MAIR(value)`: defines the otherwise unused `MemAttr7` field of the MAIR for custom tests.
 - `MAIR_DEVICE_nGnRnE (0x00)`: strongly ordered device memory.

- 5826 – MAIR_DEVICE_GRE (0x0c): standard device memory (with re-ordering, gathering and early write
- 5827 acknowledgement).
- 5828 – MAIR_NORMAL_NC (0x44): normal non-cacheable memory.
- 5829 – MAIR_NORMAL_RA_WA (0xff): normal cacheable memory.

5830 Finally, the harness allocator can be guided to place variables in locations with particular relationships
 5831 between them (in the same page or cache line, or at the same offset into their respective regions):

- 5832 ▷ INIT_REGION_OWN(*var*, *region*): that *var* owns a region of memory larger than the default of a
 5833 page, *region* can take values:
 - 5834 – REGION_OWN_CACHE_LINE: this variable only takes up a single cache line.
 - 5835 – REGION_OWN_PAGE: don't allocate other variables in the same page (the default).
 - 5836 – REGION_OWN_PMD: don't allocate other variables in the same 2MiB region.
 - 5837 – REGION_OWN_PUD: don't allocate other variables in the same 1GiB region.
- 5838 ▷ INIT_REGION_PIN(*var1*, *var2*, *region*): place *var1* and *var2* in the same region, where *region* is
 5839 one of:
 - 5840 – REGION_SAME_CACHE_LINE: place both in the same cache line.
 - 5841 – REGION_SAME_PAGE: place both in same page.
 - 5842 – REGION_SAME_PMD: place both same 2MiB region.
 - 5843 – REGION_SAME_PUD: place both same 1GiB region.
- 5844 ▷ INIT_REGION_OFFSET(*var1*, *var2*, *region*): ensure that *var1* and *var2* have the same *offset* into
 5845 the region (that is, the least significant bits overlap), where *region* can be one of:
 - 5846 – REGION_SAME_CACHE_LINE_OFFSET: ensure both have same lower CACHE_LINE_SHIFT bits.
 - 5847 – REGION_SAME_PAGE_OFFSET: ensure both have same offset into the page (bits 12-0).
 - 5848 – REGION_SAME_PMD_OFFSET: ensure both have same offset into the 2MiB region (bits 20-12).
 - 5849 – REGION_SAME_PUD_OFFSET: ensure both have same offset into the 1GiB region (bits 29-20).

Proof of virtual memory abstraction

This Appendix is based on: Relaxed virtual memory in Armv8-A [37] by Ben Simner, Alasdair Armstrong, Jean Pichon-Pharabod, Christopher Pulte, Richard Grisenthwaite, and Peter Sewell, published in the proceedings of the 31st European Symposium on Programming (ESOP, 2022). In particular, much of the proof is the work of Jean Pichon-Pharabod.

We consider a simple case when the virtual address abstraction ought to hold: under some conditions, the model with translation and the original model without translations coincide. Here, we only consider the consistency of the pre-executions, but not how these pre-executions arise.

D.1 Abstraction

Definition 1 (VA abstraction subcondition). G satisfies the *VA abstraction subcondition* when it has no page-table-affecting instructions: no TLBI, no context-changing operations (for example via writing to registers, for example via MSR TTBR), etc.

Definition 2 (VA abstraction condition). G_{tr} satisfies the *VA abstraction condition* when it satisfies the VA abstraction subcondition, and has a static injective page table.

Theorem 1 (VA abstraction). For all $(G_{tr} : \text{concrete execution})$ if G_{tr} is consistent wrt. the model with translation and respects the VA abstraction condition, then let $G_{abs} = \text{erase } G_{tr}$ in G_{abs} is consistent wrt. the model without translation.

Proof. First, the builtin `addr` of the abstract model is assumed to coincide with the derived `addr` of the concrete model by the erasure. Showing that the two definitions of pre-executions do relate in this way is outside of our scope. Given that the definitions `addr` coincide, the definitions of all the other derived relations of the abstract model, including `ob` in the translation model, are syntactically supersets of their definition in the concrete model, so a cycle in `ob` in the abstract model is also a cycle in `ob` in the concrete model. \square

D.2 Anti-abstraction

For this direction, we need to be able to put the translation table somewhere.

Step 1: Building the candidate execution in the translation model

Definition 3 (translation extension condition). The *translation extension condition* is the data of $(G_{abs} : \text{execution})$ such that G_{abs} is consistent wrt. the model without translation and has no TLBI, and no MSR TTBR and

5884 (va_space : va_address -> bool)
5885 such that all the memory accesses of Gabs are in va_space
5886 and
5887 (pt_pa_space : pa_address -> bool)
5888 (pt_initial_state : pa_address -> option (list byte)),
5889 such that the domains of pt_pa_space and pt_initial_state coincide
5890 and
5891 (tr_ctxt : translation_context),
5892 such that id_map_lifted va_space and pt_pa_space are disjoint address spaces
5893 and
5894 (translate : translation_function),
5895 such that translating abstract_va_space translate-reads from within pt_pa_space and gives the injective
5896 map.

5897 **Definition 4** (translation extension). Given the translation extension condition, the *translation extension*
5898 *Gtr* of Gabs is constructed by:

- 5899 ▷ adding all the initial writes for the page tables,
- 5900 ▷ adding all the translate reads obtained by running the `translate` function with the `tr_ctxt`,
- 5901 ▷ adding the translate reads in `iio` between the fetch and the explicit event,
- 5902 ▷ adding `tdata` to match `addr`,
- 5903 ▷ adding `trf` from the corresponding initial writes to the translates.

5904 **Definition 5** (VA anti abstraction condition). *Gtr* satisfies the *VA anti-abstraction condition* when it is
5905 derived from a consistent execution which satisfies the VA abstraction subcondition by the translation
5906 extension.

5907 **Lemma 1** (VA abstraction condition for translation extension). If *Gtr* satisfies the *VA anti-abstraction*
5908 *condition*, then *Gtr* satisfies the VA abstraction condition.

5909 *Proof.* The translation extension does not add any extra instructions, and sets up static injective page
5910 tables. □

5911 **Lemma 2** (obtlbi-empty). If *Gtr* satisfies the VA anti-abstraction condition, then `obtlbi` is empty.

5912 *Proof.* `obtlbi` has

- 5913 ▷ `obtlbi_translate` which has
 - 5914 – `tcache1`
 - 5915 which is `[T & Stage1]`; `tfr`; `tseq1`
 - 5916 the latter is
 - 5917 `[W]`; `(maybe_TLB_barriered_by_va & ob)`; `[TLBI VA]`
 - 5918 which requires a TLBI, so it is empty
 - 5919 – `tcache2` & ...
 - 5920 which requires a TLBI, so it is empty
 - 5921 – `(tcache2; ...)` & ...
 - 5922 which requires a TLBI, so it is empty
- 5923 ▷ `[M]`; `iio^-1`; `obtlbi_translate`
- 5924 to which the same reasoning applies

5925 □

5926 Step 2: Consistency

5927 **Lemma 3.** If *Gtr* satisfies the VA anti-abstraction condition, then translation-internal is acyclic.

5928 *Proof.* `po-pa`; `[W]`; `trf` is empty
5929 because by the VA anti-abstraction condition there are no non-initial writes to page tables. □

5930 So we only need to show `external` is acyclic.

5931 **Lemma 4** (`ob-to-T`). If G satisfies the VA anti-abstraction condition, then, for all $n \geq 1$,

```
5932 imm(ob)^n; [T] ==
5933   iio
5934   | imm(ob)^(n-1); trfe
5935   | imm(ob)^(n-1); [T]; iio; [T]
5936   | imm(ob)^(n-1); [CSE]; instruction-order
5937   | imm(ob)^(n-1); po; [ERET]; instruction-order; [T]
```

5938 *Proof.* \triangleright The `addr` clause

5939 | `tdata`; `[T_f]`

5940 is empty because there are no translation failures.

5941 \triangleright `tob` does not contribute: there are no faults, and no non-initial writes to page table entries.

5942 \triangleright The first clause of `ctxob` is empty because there are no MSR TTBR. The third and fourth are also empty, because they do not end in a `[T]`.

5943 \triangleright Given a static injective mapping, the new | `(addr | data | ctrl); trfi` clause of `dob` is empty.

5944

□

5946 **Lemma 5** (`no-cycle-ob-to-init`). If G_{tr} is well-formed and consistent (in either model), then there is cycle in `ob` via the initial writes.

5948 *Proof.* By well-formedness, `wco`; `[INIT] = [INIT]`; `wco`; `[INIT]`, and `wco` is acyclic.

5949 By examination of the other edges.

□

5950 **Lemma 6** (`ob-from-T`). If G_{tr} satisfies the VA anti-abstraction condition, then

```
5951 [T]; imm(ob) ==
5952   iio
5953   | [T]; iio; [M]; po; [W]
```

5954 *Proof.* By examination of the edges.

□

5955 **Lemma 7** (`instruction-order-compress`).

5956 `instruction-order`; `[T]`; `iio`; `[M]`; `po` \subseteq `instruction-order`

5957 *Proof.* If we unfold the definitions of `instruction-order` and `po`, we have

5958 `iio`⁻¹; `fpo`; `iio`; `[T]`; `iio`; `[M]`; `[M|F|C]`; `iio`⁻¹; `fpo`; `iio`; `[M|F|C]`

5959 which we can simplify into

5960 `iio`⁻¹; `fpo`; `fpo`; `iio`; `[M|F|C]`

5961 which means we have

5962 `instruction-order`.

□

5963 **Lemma 8** (`instruction-order-compress-iio`). `instruction-order`; `iio`; `po` \subseteq `instruction-order`

5964 *Proof.* `iio` is transitive, and is the RHS of `instruction-order`.

□

5965 **Lemma 9** (`ob-acyclic-preserved`). If G satisfies the VA anti-abstraction condition, if there is a cycle in `translate-ob`, then there is a cycle in `plain-ob`.

5967 *Proof.* Consider a minimal cycle in `translate-imm(ob)` (that is, the transitive closure of the `ob` of the model with translation). Let n be its length.

5968 We show that there is a cycle in `plain-ob`.

5969 Assume, for contradiction, that the cycle contains an edge that is not in `plain-ob` (that is, the `ob` of the model without translation):

5972 \triangleright `iio`
5973 by case split:

- 5974 – `[T]; iio; [M]`: by Lemma `ob-to-T`, the `ob` edge to the left has to be either
 - 5975 * `iio` in which case, by transitivity of `iio`, there is a shorter cycle, so we have a contradiction.
5976 Let us call this Case `IIOtrans`.
 - 5977 * `trfe`, which is from an initial write by the VA abstraction condition,
5978 but by Lemma `no-cycle-ob-to-init`, the cycle cannot exist.
 - 5979 * `imm(ob)^(n-2); [T]; iio; [T]; iio; [M]`
5980 then we have `imm(ob)^(n-2); [T]; iio; [M]`, which involves one fewer translate,
5981 so we have a contradiction.
 - 5982 * `imm(ob)^(n-2); [CSE]; instruction-order`
5983 This is similar to `IIOtrans`.
 - 5984 * `imm(ob)^(n-2); po; [ERET]; instruction-order; [T]`
5985 This is similar to `IIOtrans`.
- 5986 – `[T]; iio; [T]`:
5987 So the whole cycle looks like `imm(ob)^(n-1); [T]; iio; [T]`
5988 By Lemma `ob-to-T`, we have either
 - 5989 * `imm(ob)^(n-2); iio; [T]; iio; [T]`
5990 See Case `IIOtrans`.
 - 5991 * `imm(ob)^(n-2); trfe`
5992 the `trfe` is from an initial write by the VA abstraction condition,
5993 and by Lemma `no-cycle-ob-to-init`, the cycle cannot exist.
 - 5994 * `imm(ob)^(n-2); [T]; iio; [T]`
5995 but we already have `iio` to the second `T`,
5996 so we have a cycle involving one fewer translate,
5997 so we have a contradiction.
 - 5998 * `imm(ob)^(n-2); [CSE]; instruction-order`
5999 This is similar to `IIOtrans`.
 - 6000 * `imm(ob)^(n-2); po; [ERET]; instruction-order; [T]`
6001 This is similar to `IIOtrans`.

6002 \triangleright `tob` has

- 6003 – `[T_f]; tfr`
6004 which has a fault, so we have a contradiction.
- 6005 – `([T_f]; tfri) & (po; [dsb.sy]; instruction-order)^-1`
6006 which has a fault, so we have a contradiction.
- 6007 – `speculative; trfi` which is empty, because of the static page table.

6008 \triangleright `obtlbi`, which is empty by Lemma `obtlbi-empty`.

6009 \triangleright `ctxob` has

- 6010 – `speculative; [MSR TTBR]`
6011 by the VA abstraction condition, there is no `MSR TTBR`
- 6012 – `[CSE]; instruction-order`
6013 So the whole cycle looks like
6014 `[CSE]; instruction-order; imm(ob)^(n-1)`
6015 Because `instruction-order` is acyclic, $n \geq 1$, so we have
6016 `[CSE]; instruction-order; imm(ob); imm(ob)^(n-2)`
6017 By Lemma `ob-from-T`, we have either:

6018 * [CSE]; instruction-order; iio; imm(ob)⁽ⁿ⁻²⁾
6019 which means that by Lemma instruction-order-compress, we have
6020 [CSE]; instruction-order; imm(ob)⁽ⁿ⁻²⁾
6021 so we have a cycle involving one edge fewer, so we have a contradiction.

6022 * [CSE]; instruction-order; [T]; iio; [M]; po; [W]; imm(ob)⁽ⁿ⁻²⁾
6023 which means that by Lemma instruction-order-compress, we have
6024 [CSE]; instruction-order; imm(ob)⁽ⁿ⁻²⁾
6025 so we have a cycle involving one edge fewer, so we have a contradiction.

6026 – [ContextChange]; po; [CSE]
6027 by the VA abstraction condition, there is no ContextChange.

6028 – speculative; [CSE]
6029 The CSE has to be an ISB, because there are no exceptions, and the speculative is either in
6030 dob in the plain model, so we have a contradiction, or in [T]; instruction-order.
6031 So the whole cycle looks like imm(ob)⁽ⁿ⁻¹⁾; [T]; iio; [M]; po; [ISB]

6032 Because po | iio is acyclic, $n - 1$ has to be ≥ 1 , so by Lemma ob-to-T, we have either

6033 * imm(ob)⁽ⁿ⁻²⁾; iio; [T]; iio; [M]; po; [ISB]
6034 See Case IIOtrans.

6035 * trfe, which is from an initial write by the VA abstraction condition,
6036 but by Lemma no-cycle-ob-to-init, the cycle cannot exist

6037 * imm(ob)⁽ⁿ⁻²⁾; [T]; iio; [T]; iio; [M]; po; [ISB]
6038 but we already have iio to the second T,
6039 so we have a cycle involving one fewer translate,
6040 so we have a contradiction.

6041 * imm(ob)⁽ⁿ⁻²⁾; [CSE]; instruction-order; [T]; iio; [M]; po; [ISB]
6042 which means that by Lemma instruction-order-compress, we have
6043 imm(ob)⁽ⁿ⁻²⁾; [CSE]; instruction-order
6044 so we have a cycle involving one edge fewer,
6045 so we have a contradiction.

6046 * imm(ob)⁽ⁿ⁻²⁾; po; [ERET]; instruction-order; [T]; iio; [M]; po; [ISB]
6047 is similar

6048 – po; [ERET]; instruction-order; [T]
6049 So the whole cycle looks like
6050 po; [ERET]; instruction-order; [T]; imm(ob)⁽ⁿ⁻¹⁾
6051 Because instruction-order is acyclic, $n \geq 1$, so we have
6052 po; [ERET]; instruction-order; [T]; imm(ob); imm(ob)⁽ⁿ⁻²⁾
6053 By Lemma ob-from-T, we have either:

6054 * po; [ERET]; instruction-order; [T]; iio; imm(ob)⁽ⁿ⁻²⁾
6055 which means that by Lemma instruction-order-compress-iio, we have
6056 po; [ERET]; instruction-order; imm(ob)⁽ⁿ⁻²⁾
6057 so we have a cycle involving one edge fewer, so we have a contradiction.

6058 * po; [ERET]; instruction-order; [T]; ([T]; iio; [M]; po; [W]); imm(ob)⁽ⁿ⁻²⁾
6059 which means that by Lemma instruction-order-compress, we have
6060 po; [ERET]; instruction-order; imm(ob)⁽ⁿ⁻²⁾
6061 so we have a cycle involving one edge fewer, so we have a contradiction.

6062 ▷ extended dob:

6063 – involving trfi from non-initial writes, which contradicts our assumption about static translation.

6064 – or [T]; instruction-order; [W],
6065 so [T]; iio; [M]; po; [W]
6066 So the whole cycle looks like imm(ob)⁽ⁿ⁻¹⁾; [T]; iio; [M]; po; [W]

6067 Because po | iio is acyclic, $n - 1$ has to be ≥ 1 , so by Lemma ob-to-T, we have either

6068 * $\text{imm}(\text{ob})^{(n-2)}$; iio ; $[\text{T}]$; iio ; $[\text{M}]$; po ; $[\text{W}]$
6069 See Case IIOtrans .

6070 * trfe , which is from an initial write by the VA abstraction condition,
6071 but by Lemma $\text{no-cycle-ob-to-init}$, the cycle cannot exist

6072 * $\text{imm}(\text{ob})^{(n-2)}$; $[\text{T}]$; iio ; $[\text{T}]$; iio ; $[\text{M}]$; po ; $[\text{W}]$
6073 but we already have iio to the second T ,
6074 so we have a cycle involving one fewer translate,
6075 so we have a contradiction.

6076 * $\text{imm}(\text{ob})^{(n-2)}$; $[\text{CSE}]$; instruction-order ; $[\text{T}]$; iio ; $[\text{M}]$; po ; $[\text{W}]$
6077 which means that by Lemma $\text{instruction-order-compress}$, we have
6078 $\text{imm}(\text{ob})^{(n-2)}$; $[\text{CSE}]$; instruction-order
6079 so we have a cycle involving one edge fewer,
6080 so we have a contradiction.

6081 * $\text{imm}(\text{ob})^{(n-2)}$; po ; $[\text{ERET}]$; instruction-order ; $[\text{T}]$; iio ; $[\text{M}]$; po ; $[\text{W}]$
6082 is similar

6083 \triangleright extended bob , but only involving TLBI , which contradicts our assumption of no TLBI .

6084 \triangleright extended obs , but only involving trfe , by the VA abstraction condition, the only writes to page
6085 tables are from initial writes, and by Lemma $\text{no-cycle-ob-to-init}$, there are no ob cycles via initial
6086 writes, so there is no cycle.

6087 \triangleright obfault , which involves a fault, which contradicts our assumptions.

6088 \triangleright obets , which involves a fault or a TLBI , which contradicts our assumptions.

6089 All the other edges are in plain-ob by definition. □

6090 **Theorem 2** (VA anti-abstraction). If the translation extension condition holds, then there exists a Gtr that
6091 satisfies the VA anti-abstraction condition such that Gtr is a stitching of Gabs with the pt_initial_state
6092 according to translate in tr_ctxt and Gtr is consistent wrt. the model with translation.

6093 *Proof.* Gtr exists by the translation extension construction,
6094 and it is consistent by Lemma $\text{ob-acyclic-preserved}$. □

Bibliography

6095

- [1] Intel Corporation. Intel 64 and IA-32 architectures software developer’s manual combined volumes: 1, 2a, 2b, 2c, 2d, 3a, 3b, 3c, 3d and 4. <https://software.intel.com/en-us/download/intel-64-and-ia-32-architectures-sdm-combined-volumes-1-2a-2b-2c-2d-3a-3b-3c-3d-and-4>, accessed 2019-06-30, May 2019. 325462-070US.
- [2] Susmit Sarkar, Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Tom Ridge, Thomas Braibant, Magnus Myreen, and Jade Alglave. The semantics of x86-CC multiprocessor machine code. In *Proceedings of POPL 2009: the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, pages 379–391, January 2009. doi:10.1145/1594834.1480929.
- [3] Scott Owens, Susmit Sarkar, and Peter Sewell. A better x86 memory model: x86-TSO. In *Proceedings of TPHOLs 2009: Theorem Proving in Higher Order Logics, LNCS 5674*, pages 391–407, 2009. doi:10.1007/978-3-642-03359-9_27.
- [4] Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. x86-TSO: A rigorous and usable programmer’s model for x86 multiprocessors. *Communications of the ACM*, 53(7):89–97, July 2010. (Research Highlights). doi:10.1145/1785414.1785443.
- [5] Anthony C. J. Fox and Magnus O. Myreen. A trustworthy monadic formalization of the ARMv7 instruction set architecture. In *Interactive Theorem Proving, First International Conference, ITP 2010, Edinburgh, UK, July 11-14, 2010. Proceedings*, pages 243–258, 2010. doi:10.1007/978-3-642-14052-5_18.
- [6] Christopher Pulte. *The Semantics of Multicopy Atomic ARMv8 and RISC-V*. PhD thesis, University of Cambridge, 2019. <https://doi.org/10.17863/CAM.39379>.
- [7] Christopher Pulte, Shaked Flur, Will Deacon, Jon French, Susmit Sarkar, and Peter Sewell. Simplifying ARM Concurrency: Multicopy-atomic Axiomatic and Operational Models for ARMv8. In *Proceedings of the 45th ACM SIGPLAN Symposium on Principles of Programming Languages*, January 2018. doi:10.1145/3158107.
- [8] Susmit Sarkar, Peter Sewell, Jade Alglave, Luc Maranget, and Derek Williams. Understanding POWER multiprocessors. In *Proceedings of PLDI 2011: the 32nd ACM SIGPLAN conference on Programming Language Design and Implementation*, pages 175–186, 2011. doi:10.1145/1993498.1993520.
- [9] The RISC-V Foundation. Sail RISC-V model. <https://github.com/riscv/sail-riscv> (accessed 2023-05-24).
- [10] Alastair Reid. Trustworthy specifications of ARM v8-A and v8-M system level architecture. In *FMCAD 2016*, pages 161–168, October 2016. URL: <https://alastairreid.github.io/papers/fmcad2016-trustworthy.pdf>, doi:10.1109/FMCAD.2016.7886675.
- [11] Alastair Reid. ARM releases machine readable architecture specification. <https://alastairreid.github.io/ARM-v8a-xml-release/>, April 2017.
- [12] Arm Limited. Arm architecture reference manual. Armv8, for Armv8-A architecture profile. <https://developer.arm.com/documentation/ddi0487/ha/?lang=en>, February 2022. H.a Armv9 EAC. ARM DDI 0487H.a (ID020222). 11530pp.
- [13] William W. Collier. *Reasoning About Parallel Architectures*. Prentice Hall, Upper Saddle River, NJ, USA, 1992.

- 6136 [14] Jade Alglave, Anthony Fox, Samin Ishtiaq, Magnus O. Myreen, Susmit Sarkar, Peter Sewell, and
6137 Francesco Zappa Nardelli. The semantics of Power and ARM multiprocessor machine code. In
6138 *Proc. DAMP 2009*, January 2009. doi:[10.1145/1481839.1481842](https://doi.org/10.1145/1481839.1481842).
- 6139 [15] Kathryn E. Gray, Gabriel Kerneis, Dominic Mulligan, Christopher Pulte, Susmit Sarkar, and Peter
6140 Sewell. An integrated concurrency and core-ISA architectural envelope definition, and test oracle,
6141 for IBM POWER multiprocessors. In *Proc. MICRO-48, the 48th Annual IEEE/ACM International
6142 Symposium on Microarchitecture*, December 2015. doi:[10.1145/2830772.2830775](https://doi.org/10.1145/2830772.2830775).
- 6143 [16] Shaked Flur, Susmit Sarkar, Christopher Pulte, Kyndylan Nienhuis, Luc Maranget, Kathryn E.
6144 Gray, Ali Sezgin, Mark Batty, and Peter Sewell. Mixed-size concurrency: ARM, POWER, C/C++11,
6145 and SC. In *The 44th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming
6146 Languages, Paris, France*, pages 429–442, January 2017. doi:[10.1145/3009837.3009839](https://doi.org/10.1145/3009837.3009839).
- 6147 [17] Azalea Raad and Viktor Vafeiadis. Persistence semantics for weak memory: Integrating epoch
6148 persistency with the tso memory model. *Proc. ACM Program. Lang.*, 2(OOPSLA), October 2018.
6149 doi:[10.1145/3276507](https://doi.org/10.1145/3276507).
- 6150 [18] P. Cenciarelli, A. Knapp, and E. Sibilio. The Java memory model: Operationally, denotationally,
6151 axiomatically. In *ESOP*, 2007. doi:[10.1007/978-3-540-71316-6_23](https://doi.org/10.1007/978-3-540-71316-6_23).
- 6152 [19] Mark Batty, Kayvan Memarian, Scott Owens, Susmit Sarkar, and Peter Sewell. Clarifying and
6153 Compiling C/C++ Concurrency: from C++11 to POWER. In *Proceedings of POPL 2012: The
6154 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (Philadelphia)*,
6155 pages 509–520, 2012. doi:[10.1145/2103656.2103717](https://doi.org/10.1145/2103656.2103717).
- 6156 [20] Susmit Sarkar, Kayvan Memarian, Scott Owens, Mark Batty, Peter Sewell, Luc Maranget, Jade
6157 Alglave, and Derek Williams. Synchronising C/C++ and POWER. In *Proceedings of PLDI 2012, the
6158 33rd ACM SIGPLAN conference on Programming Language Design and Implementation (Beijing)*,
6159 pages 311–322, 2012. doi:[10.1145/2254064.2254102](https://doi.org/10.1145/2254064.2254102).
- 6160 [21] Andrew S. Tanenbaum and Herbert Bos. *Modern Operating Systems*. Prentice Hall Press, USA, 4th
6161 edition, 2014.
- 6162 [22] Ross J. Anderson. *Security Engineering: A Guide to Building Dependable Distributed Systems*. John
6163 Wiley & Sons, Inc., USA, 3rd edition, 2021.
- 6164 [23] S.V. Adve and K. Gharachorloo. Shared memory consistency models: a tutorial. *Computer*,
6165 29(12):66–76, 1996. doi:[10.1109/2.546611](https://doi.org/10.1109/2.546611).
- 6166 [24] William Pugh. The Java memory model is fatally flawed. *Concurrency: Practice and Experience*,
6167 12(6):445–455, 2000. doi:[https://doi.org/10.1002/1096-9128\(200005\)12:6<445::AID-CPE484>3.0.CO;2-A](https://doi.org/10.1002/1096-9128(200005)12:6<445::AID-CPE484>3.0.CO;2-A).
- 6168
- 6169 [25] RISC-V Foundation. The RISC-V Instruction Set Manual, Volume I: User-Level ISA, doc-
6170 ument version 20191213. [https://github.com/riscv/riscv-isa-manual/releases/download/
6171 Ratified-IMAFDQC/riscv-spec-20191213.pdf](https://github.com/riscv/riscv-isa-manual/releases/download/Ratified-IMAFDQC/riscv-spec-20191213.pdf), December 2019.
- 6172 [26] *Power ISATM Version 2.07*. IBM, 2013.
- 6173 [27] P. Becker, editor. *Programming Languages — C++*. 2011. ISO/IEC 14882:2011. [http://www.
6174 open-std.org/jtc1/sc22/wg21/docs/papers/2011/n3242.pdf](http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2011/n3242.pdf).
- 6175 [28] Mark John Batty. *The C11 and C++11 Concurrency Model*. PhD thesis, University of Cambridge,
6176 2014. 2015 SIGPLAN John C. Reynolds Doctoral Dissertation award and 2015 CPHC/BCS
6177 Distinguished Dissertation Competition winner.
- 6178 [29] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, Alex Buckley, Daniel Smith, and Gavin Bierman.
6179 The Java Language Specification. <https://docs.oracle.com/javase/specs/jls/se20/jls20.pdf>,
6180 March 2023. JSR-395.
- 6181 [30] std::sync::atomic - Rust. <https://doc.rust-lang.org/std/sync/atomic/index.html> (accessed
6182 2023-05-30).
- 6183 [31] Data.IOREf - hackage.haskell.org. [https://hackage.haskell.org/package/base-4.18.0.0/docs/
6184 Data-IORef.html](https://hackage.haskell.org/package/base-4.18.0.0/docs/Data-IORef.html) (accessed 2023-05-30).

- 6185 [32] Jade Alglave, Luc Maranget, and Michael Tautschnig. Herding Cats: Modelling, Simulation,
6186 Testing, and Data Mining for Weak Memory. *ACM TOPLAS*, 36(2):7:1–7:74, July 2014. doi:
6187 [10.1145/2627752](https://doi.org/10.1145/2627752).
- 6188 [33] Jade Alglave, Luc Maranget, Kate Deplaix, Keryan Didier, and Susmit Sarkar. The litmus7 tool.
6189 <http://diy.inria.fr/doc/litmus.html/>, 2019. Accessed 2019-07-08.
- 6190 [34] Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell. Fences in weak memory models. In
6191 *Proc. CAV*, 2010. doi:[10.1007/978-3-642-14295-6_25](https://doi.org/10.1007/978-3-642-14295-6_25).
- 6192 [35] Ben Simner, Shaked Flur, Christopher Pulte, Alasdair Armstrong, Jean Pichon-Pharabod, Luc
6193 Maranget, and Peter Sewell. ARMv8-A system semantics: instruction fetch in relaxed architectures.
6194 In *Proceedings of the 29th European Symposium on Programming, ESOP 2020*, 2020. doi:[10.1007/](https://doi.org/10.1007/978-3-030-44914-8_23)
6195 [978-3-030-44914-8_23](https://doi.org/10.1007/978-3-030-44914-8_23).
- 6196 [36] Alasdair Armstrong, Brian Campbell, Ben Simner, Christopher Pulte, and Peter Sewell. Isla:
6197 Integrating full-scale ISA semantics and axiomatic concurrency models. In *Proceedings of the 33rd*
6198 *International Conference on Computer Aided Verification, CAV 2021*, July 2021. doi:[10.1007/](https://doi.org/10.1007/978-3-030-81685-8_14)
6199 [978-3-030-81685-8_14](https://doi.org/10.1007/978-3-030-81685-8_14).
- 6200 [37] Ben Simner, Alasdair Armstrong, Jean Pichon-Pharabod, Christopher Pulte, Richard Grisenthwaite,
6201 and Peter Sewell. Relaxed virtual memory in Armv8-A. In *Proceedings of the 31st European*
6202 *Symposium on Programming, ESOP 2022*, April 2022. doi:[10.1007/978-3-030-99336-8_6](https://doi.org/10.1007/978-3-030-99336-8_6).
- 6203 [38] Alasdair Armstrong, Brian Campbell, Ben Simner, Christopher Pulte, and Peter Sewell. Isla:
6204 Integrating full-scale ISA semantics and axiomatic concurrency models (extended version). *Formal*
6205 *Methods in System Design*, 63(1):110–133, October 2024. doi:[10.1007/s10703-023-00409-y](https://doi.org/10.1007/s10703-023-00409-y).
- 6206 [39] Ben Simner, Alasdair Armstrong, Thomas Baueriss, Brian Campbell, Ohad Kammar, Jean Pichon-
6207 Pharabod, and Peter Sewell. Precise exceptions in relaxed architectures (pre-publication), December
6208 2024. URL: <https://arxiv.org/abs/2412.15140>.
- 6209 [40] Luc Maranget, Susmit Sarkar, and Peter Sewell. A tutorial introduction to the ARM and
6210 POWER relaxed memory models, October 2012. Draft. URL: [https://www.cl.cam.ac.uk/~pes20/](https://www.cl.cam.ac.uk/~pes20/ppc-supplemental/test7.pdf)
6211 [ppc-supplemental/test7.pdf](https://www.cl.cam.ac.uk/~pes20/ppc-supplemental/test7.pdf).
- 6212 [41] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs.
6213 *IEEE Trans. Comput.*, C-28(9):690–691, 1979.
- 6214 [42] Jade Alglave, Will Deacon, Richard Grisenthwaite, Antoine Hacquard, and Luc Maranget. Armed
6215 Cats: Formal Concurrency Modelling at Arm. *ACM Trans. Program. Lang. Syst.*, 43(2), July 2021.
6216 doi:[10.1145/3458926](https://doi.org/10.1145/3458926).
- 6217 [43] Mustaque Ahamad, Gil Neiger, James E. Burns, Prince Kohli, and Phillip W. Hutto. Causal
6218 memory: definitions, implementation, and programming. *Distributed Computing*, 9(1):37–49, 1995.
6219 URL: <http://dx.doi.org/10.1007/BF01784241>, doi:[10.1007/BF01784241](https://doi.org/10.1007/BF01784241).
- 6220 [44] Arm Limited. Exploration Tools – Arm Developer. [https://developer.arm.com/downloads/-/](https://developer.arm.com/downloads/-/exploration-tools)
6221 [exploration-tools](https://developer.arm.com/downloads/-/exploration-tools), 2022. Accessed May 2022.
- 6222 [45] Alasdair Armstrong, Thomas Bauereiss, Brian Campbell, Alastair Reid, Kathryn E. Gray, Robert M.
6223 Norton, Prashanth Mundkur, Mark Wassell, Jon French, Christopher Pulte, Shaked Flur, Ian Stark,
6224 Neel Krishnaswami, and Peter Sewell. ISA semantics for ARMv8-A, RISC-V, and CHERI-MIPS. In
6225 *Proc. 46th ACM SIGPLAN Symposium on Principles of Programming Languages*, January 2019.
6226 Proc. ACM Program. Lang. 3, POPL, Article 71. doi:[10.1145/3290384](https://doi.org/10.1145/3290384).
- 6227 [46] Alasdair Armstrong, Brian Campbell, Ben Simner, Christopher Pulte, and Peter Sewell. Isla:
6228 Integrating full-scale ISA semantics and axiomatic concurrency models. In *In Proc. 33rd International*
6229 *Conference on Computer-Aided Verification*, July 2021. Extended version available at <https://www.cl.cam.ac.uk/~pes20/isla/isla-cav2021-extended.pdf>.
6230
- 6231 [47] Shaked Flur. *Thesis (pre-publication)*. PhD thesis, University of Cambridge, January 2024.
- 6232 [48] Shaked Flur, Kathryn E. Gray, Christopher Pulte, Susmit Sarkar, Ali Sezgin, Luc Maranget, Will
6233 Deacon, and Peter Sewell. Modelling the ARMv8 Architecture, Operationally: Concurrency and

- 6234 ISA. In *Proceedings of POPL: the 43rd ACM SIGPLAN-SIGACT Symposium on Principles of*
6235 *Programming Languages*, 2016. doi:10.1145/2914770.2837615.
- 6236 [49] Scott Owens, Peter Böhm, Francesco Zappa Nardelli, and Peter Sewell. Lem: A lightweight tool
6237 for heavyweight semantics. In *Proceedings of Interactive Theorem Proving – Second International*
6238 *Conference (previously TPHOLs) (Berg en Dal), LNCS 6898*, pages 363–369, 2011. (Rough Diamond).
6239 doi:10.1007/978-3-642-22863-6_27.
- 6240 [50] Dominic P. Mulligan, Scott Owens, Kathryn E. Gray, Tom Ridge, and Peter Sewell. Lem: Reusable
6241 engineering of real-world semantics. In *Proceedings of the 19th ACM SIGPLAN International*
6242 *Conference on Functional Programming, ICFP '14*, pages 175–188, New York, NY, USA, September
6243 2014. ACM. doi:10.1145/2628136.2628143.
- 6244 [51] Shaked Flur, Jon French, Kathryn Gray, Christopher Pulte, Susmit Sarkar, and Peter Sewell. rmem.
6245 www.cl.cam.ac.uk/~pes20/rmem/, 2017.
- 6246 [52] Jeremy Manson, William Pugh, and Sarita V. Adve. The Java memory model. In *Proceedings of*
6247 *the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL*
6248 *'05*, page 378391, New York, NY, USA, 2005. Association for Computing Machinery. doi:10.1145/
6249 1040305.1040336.
- 6250 [53] Will Deacon. The ARMv8 application level memory model. [https://github.com/herd/herdtools7/
6251 blob/master/herd/libdir/aarch64.cat](https://github.com/herd/herdtools7/blob/master/herd/libdir/aarch64.cat) (accessed 2019-07-01), 2016.
- 6252 [54] Arm. The Armv9 application level memory model. [https://github.com/herd/herdtools7/blob/
6253 master/herd/libdir/aarch64.cat](https://github.com/herd/herdtools7/blob/master/herd/libdir/aarch64.cat) (accessed 2023-12-21), 2016.
- 6254 [55] Gerwin Klein, June Andronick, Kevin Elphinstone, Toby Murray, Thomas Sewell, Rafal Kolanski,
6255 and Gernot Heiser. Comprehensive formal verification of an OS microkernel. *ACM Transactions on*
6256 *Computer Systems*, 32(1):2:1–2:70, February 2014. doi:10.1145/2560537.
- 6257 [56] Ronghui Gu, Zhong Shao, Jieung Kim, Xiongnan (Newman) Wu, Jérémie Koenig, Vilhelm Sjöberg,
6258 Hao Chen, David Costanzo, and Tahina Ramananandro. Certified concurrent abstraction layers.
6259 In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and*
6260 *Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, pages 646–661, 2018.
6261 doi:10.1145/3192366.3192381.
- 6262 [57] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan (Newman) Wu, Jieung Kim, Vilhelm Sjöberg,
6263 and David Costanzo. CertiKOS: An extensible architecture for building certified concurrent
6264 OS kernels. In *12th USENIX Symposium on Operating Systems Design and Implementation,*
6265 *OSDI 2016, Savannah, GA, USA, November 2-4, 2016.*, pages 653–669, 2016. URL: [https:
6266 //www.usenix.org/conference/osdi16/technical-sessions/presentation/gu](https://www.usenix.org/conference/osdi16/technical-sessions/presentation/gu).
- 6267 [58] Andrew Ferraiuolo, Andrew Baumann, Chris Hawblitzel, and Bryan Parno. Komodo: Using
6268 verification to disentangle secure-enclave hardware from software. In *Proceedings of the 26th*
6269 *Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*, pages 287–305,
6270 2017. doi:10.1145/3132747.3132782.
- 6271 [59] Roberto Guanciale, Hamed Nemati, Mads Dam, and Christoph Baumann. Provably secure memory
6272 isolation for linux on ARM. *J. Comput. Secur.*, 24(6):793–837, 2016. doi:10.3233/JCS-160558.
- 6273 [60] Christoph Baumann, Oliver Schwarz, and Mads Dam. Compositional verification of security
6274 properties for embedded execution platforms. In *PROOFS@CHES 2017, 6th International Workshop*
6275 *on Security Proofs for Embedded Systems, Taipei, Taiwan, Friday September 29th, 2017*, pages 1–16,
6276 2017. URL: <http://www.easychair.org/publications/paper/wkpS>.
- 6277 [61] Yong Kiam Tan, Magnus O. Myreen, Ramana Kumar, Anthony C. J. Fox, Scott Owens, and
6278 Michael Norrish. The verified CakeML compiler backend. *J. Funct. Program.*, 29:e2, 2019. doi:
6279 10.1017/S0956796818000229.
- 6280 [62] Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. CakeML: a verified
6281 implementation of ML. In *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of*
6282 *Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, pages 179–192,
6283 2014. doi:10.1145/2535838.2535841.

- 6284 [63] Xavier Leroy. A formally verified compiler back-end. *J. Autom. Reasoning*, 43(4):363–446, 2009.
6285 [doi:10.1007/s10817-009-9155-4](https://doi.org/10.1007/s10817-009-9155-4).
- 6286 [64] Hongxu Cai, Zhong Shao, and Alexander Vaynberg. Certified self-modifying code. *SIGPLAN Not.*,
6287 42(6):6677, June 2007. [doi:10.1145/1273442.1250743](https://doi.org/10.1145/1273442.1250743).
- 6288 [65] Magnus O. Myreen. Verified just-in-time compiler on x86. In *Proceedings of the 37th Annual ACM*
6289 *SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '10, pages 107–118,
6290 New York, NY, USA, 2010. ACM. [doi:10.1145/1706299.1706313](https://doi.org/10.1145/1706299.1706313).
- 6291 [66] Arm Limited. Arm architecture reference manual. Armv8, for Armv8-A architecture profile. [https://
6292 developer.arm.com/documentation/ddi0487/da/?lang=en](https://developer.arm.com/documentation/ddi0487/da/?lang=en), October 2017. D.a Armv8.4 EAC.
6293 ARM DDI 0487D.a (ID103018). 7476pp.
- 6294 [67] Richard Grisenthwaite. personal communication, 2020.
- 6295 [68] Arm Limited. Arm architecture reference manual. [https://developer.arm.com/documentation/
6296 ddi0487/ka/?lang=en](https://developer.arm.com/documentation/ddi0487/ka/?lang=en), March 2024. K.a Armv9 EAC. ARM DDI 0487K.a (ID032224). 14777pp.
- 6297 [69] Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell. Litmus: running tests against
6298 hardware. In *Proceedings of TACAS 2011: the 17th international conference on Tools and Algorithms*
6299 *for the Construction and Analysis of Systems*, pages 41–44, Berlin, Heidelberg, 2011. Springer-Verlag.
6300 URL: <http://dl.acm.org/citation.cfm?id=1987389.1987395>, [doi:10.1007/978-3-642-19835-9_](https://doi.org/10.1007/978-3-642-19835-9_5)
6301 [5](https://doi.org/10.1007/978-3-642-19835-9_5).
- 6302 [70] Jade Alglave and Luc Maranget. The diy7 tool. <http://diy.inria.fr/>, 2019. Accessed 2021-07-01.
- 6303 [71] Hrutvik Kanabar, Anthony C. J. Fox, and Magnus O. Myreen. Taming an Authoritative Armv8 ISA
6304 Specification: L3 Validation and CakeML Compiler Verification. In June Andronick and Leonardo
6305 de Moura, editors, *13th International Conference on Interactive Theorem Proving (ITP 2022)*,
6306 volume 237 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 20:1–20:22, Dagstuhl,
6307 Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. URL: [https://drops.dagstuhl.
6308 de/entities/document/10.4230/LIPIcs.ITP.2022.20](https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ITP.2022.20), [doi:10.4230/LIPIcs.ITP.2022.20](https://doi.org/10.4230/LIPIcs.ITP.2022.20).
- 6309 [72] Alastair Reid, Rick Chen, Anastasios Deligiannis, David Gilday, David Hoyes, Will Keen, Ashan
6310 Pathirane, Owen Shepherd, Peter Vrabel, and Ali Zaidi. End-to-end verification of processors with
6311 ISA-Formal. In Swarat Chaudhuri and Azadeh Farzan, editors, *Computer Aided Verification - 28th*
6312 *International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II*,
6313 volume 9780 of *Lecture Notes in Computer Science*, pages 42–58. Springer, 2016.
- 6314 [73] Arm Limited. Arm Cortex-A53 MPCore Processor Technical Reference Manual, 2022. ARM DDI
6315 0500J.
- 6316 [74] John L. Hennessy and David A. Patterson. *Computer Architecture, Sixth Edition: A Quantitative*
6317 *Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 6th edition, 2017.
- 6318 [75] Arm Limited. Arm architecture reference manual. Armv8, for Armv8-A architecture profile. [https://
6319 developer.arm.com/documentation/ddi0487/ea/?lang=en](https://developer.arm.com/documentation/ddi0487/ea/?lang=en), July 2019. E.a Armv8.5 EAC. ARM
6320 DDI 0487E.a (ID070919). 7900pp.
- 6321 [76] Arm Limited. Arm architecture reference manual. Armv8, for Armv8-A architecture profile. [https://
6322 developer.arm.com/documentation/ddi0487/gb/?lang=en](https://developer.arm.com/documentation/ddi0487/gb/?lang=en), 2021. G.b Armv8.7 EAC. ARM DDI
6323 0487G.b (ID072021). 8696pp.
- 6324 [77] Arm Limited. Arm architecture reference manual. Armv8, for Armv8-A architecture profile. [https://
6325 developer.arm.com/documentation/ddi0487/fa/?lang=en](https://developer.arm.com/documentation/ddi0487/fa/?lang=en), February 2020. F.a Armv8.6 Beta.
6326 ARM DDI 0487F.a (ID021920). 8128pp.
- 6327 [78] The Chromium source code. Jump table assembler - src/wasm/jump-table-assembler.cc - v8/v8.git.
- 6328 [79] The Chromium source code. Commit ea82d: [arm64] Use BTI instructions for forward CFI.
- 6329 [80] Jim Handy. *The cache memory book*. Academic Press Professional, Inc., 1993.
- 6330 [81] Arm Limited. Arm architecture reference manual. [https://developer.arm.com/documentation/
6331 ddi0487/ja/?lang=en](https://developer.arm.com/documentation/ddi0487/ja/?lang=en), April 2023. J.a Armv9 EAC. ARM DDI 0487J.a (ID042523). 12940pp.

- 6332 [82] Hongxu Cai, Zhong Shao, and Alexander Vaynberg. Certified self-modifying code. In *Proceedings*
6333 *of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*,
6334 PLDI '07, page 6677, New York, NY, USA, 2007. Association for Computing Machinery. doi:
6335 [10.1145/1250734.1250743](https://doi.org/10.1145/1250734.1250743).
- 6336 [83] Thomas Dinsdale-Young, Mike Dodds, Philippa Gardner, Matthew J. Parkinson, and Viktor
6337 Vafeiadis. Concurrent abstract predicates. In Theo D'Hondt, editor, *ECOOP 2010 - Object-*
6338 *Oriented Programming, 24th European Conference, Maribor, Slovenia, June 21-25, 2010. Proceedings*,
6339 volume 6183 of *Lecture Notes in Computer Science*, pages 504–528. Springer, 2010. doi:[10.1007/](https://doi.org/10.1007/978-3-642-14107-2_24)
6340 [978-3-642-14107-2_24](https://doi.org/10.1007/978-3-642-14107-2_24).
- 6341 [84] Shilpi Goel and Warren A. Hunt. Automated code proofs on a formal model of the X86. In Ernie
6342 Cohen and Andrey Rybalchenko, editors, *Verified Software: Theories, Tools, Experiments*, pages
6343 222–241, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg. doi:[10.1007/978-3-642-54108-7_12](https://doi.org/10.1007/978-3-642-54108-7_12).
- 6344 [85] Shilpi Goel, Warren A. Hunt, Matt Kaufmann, and Soumava Ghosh. Simulation and formal
6345 verification of x86 machine-code programs that make system calls. In *Proceedings of the 14th*
6346 *Conference on Formal Methods in Computer-Aided Design*, FMCAD '14, pages 18:91–18:98, Austin,
6347 TX, 2014. FMCAD Inc. doi:[10.1109/FMCAD.2014.6987600](https://doi.org/10.1109/FMCAD.2014.6987600).
- 6348 [86] Shilpi Goel, Anna Slobodova, Rob Sumners, and Sol Swords. Verifying x86 instruction implementa-
6349 tions. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs*
6350 *and Proofs*, CPP 2020, page 4760, New York, NY, USA, 2020. Association for Computing Machinery.
6351 doi:[10.1145/3372885.3373811](https://doi.org/10.1145/3372885.3373811).
- 6352 [87] Daniel Lustig, Geet Sethi, Margaret Martonosi, and Abhishek Bhattacharjee. COATCheck: Verifying
6353 memory ordering at the hardware-OS interface. *SIGOPS Oper. Syst. Rev.*, 50(2):233–247, March
6354 2016. doi:[10.1145/2954680.2872399](https://doi.org/10.1145/2954680.2872399).
- 6355 [88] S. Li, X. Li, R. Gu, J. Nieh, and J. Hui. A Secure and Formally Verified Linux KVM Hypervisor.
6356 In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 839–856, Los Alamitos, CA, USA,
6357 May 2021. IEEE Computer Society. doi:[10.1109/SP40001.2021.00049](https://doi.org/10.1109/SP40001.2021.00049).
- 6358 [89] Azalea Raad, John Wickerson, and Viktor Vafeiadis. Weak persistency semantics from the ground
6359 up: Formalising the persistency semantics of ARMv8 and transactional models. *Proc. ACM Program.*
6360 *Lang.*, 3(OOPSLA):135:1–135:27, October 2019. doi:[10.1145/3360561](https://doi.org/10.1145/3360561).
- 6361 [90] Azalea Raad, John Wickerson, Gil Neiger, and Viktor Vafeiadis. Persistency semantics of the
6362 Intel-x86 architecture. *PACMPL*, 4(POPL):11:1–11:31, 2020. doi:[10.1145/3371079](https://doi.org/10.1145/3371079).
- 6363 [91] Alasdair Armstrong, Brian Campbell, Ben Simner, Thibaut Pérami, and Peter Sewell. Isla Axiomatic
6364 Models. <https://github.com/rem-s-project/system-semantics-arm-axiomatic-models>.
- 6365 [92] Arm Limited. Arm Cortex-A57 MPCore Processor Technical Reference Manual, 2016. ARM DDI
6366 0488H.
- 6367 [93] Arm Limited. Arm Cortex-A72 MPCore Processor Technical Reference Manual, 2016. ARM 100095
6368 r0p3.
- 6369 [94] Arm Limited. Arm Cortex-A76 Core Processor Technical Reference Manual, 2023. ARM 100798
6370 r4p1.
- 6371 [95] Arm Limited. Arm Cortex-A78 Core Processor Technical Reference Manual, 2023. ARM 101430
6372 r1p2.
- 6373 [96] Arm Limited. Arm Cortex-X1 Core Processor Technical Reference Manual, 2023. ARM 101433
6374 r1p2.
- 6375 [97] Arm Limited. Arm architecture reference manual. [https://developer.arm.com/documentation/](https://developer.arm.com/documentation/ddi0487/1a/?lang=en)
6376 [ddi0487/1a/?lang=en](https://developer.arm.com/documentation/ddi0487/1a/?lang=en), November 2024. L.a Armv9 EAC. ARM DDI 0487L.a. 14568pp.
- 6377 [98] Shilpi Goel. The x86isa books: Features, usage, and future plans. In *Proceedings 14th International*
6378 *Workshop on the ACL2 Theorem Prover and its Applications, Austin, Texas, USA, May 22-23,*
6379 *2017.*, pages 1–17, 2017. <https://arxiv.org/abs/1705.01225>. doi:[10.4204/EPTCS.249.1](https://doi.org/10.4204/EPTCS.249.1).
- 6380 [99] Rishiyur S. Nikhil and Niraj Nayan Sharma. Forvis: A formal RISC-V ISA specification. [https://](https://github.com/rsnikhil/Forvis_RISCV-ISA-Spec)
6381 github.com/rsnikhil/Forvis_RISCV-ISA-Spec, 2019. Accessed 2019-07-01.

- 6382 [100] Ian J Clester, Thomas Bourgeat, Andy Wright, Samuel Gruetter, and Adam Chlipala. riscv-plv
6383 risc-v isa formal specification. <https://github.com/mit-plv/riscv- semantics>, 2019. Accessed
6384 2019-07-01.
- 6385 [101] Hira Syeda and Gerwin Klein. Reasoning about translation lookaside buffers. In *LPAR-21, 21st*
6386 *International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Maun,*
6387 *Botswana, May 7-12, 2017*, pages 490–508, 2017. doi:10.29007/c2f1.
- 6388 [102] Hira Taqdees Syeda and Gerwin Klein. Program verification in the presence of cached address
6389 translation. In *Interactive Theorem Proving - 9th International Conference, ITP 2018, Held as Part*
6390 *of the Federated Logic Conference, FloC 2018, Oxford, UK, July 9-12, 2018, Proceedings*, pages
6391 542–559, 2018. doi:10.1007/978-3-319-94821-8_32.
- 6392 [103] Bogdan F. Romanescu, Alvin R. Lebeck, and Daniel J. Sorin. Specifying and dynamically verifying
6393 address translation-aware memory consistency. In *Proceedings of the Fifteenth Edition of ASPLOS*
6394 *on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV, pages
6395 323–334, New York, NY, USA, 2010. ACM. doi:10.1145/1736020.1736057.
- 6396 [104] Bogdan Romanescu, Alvin Lebeck, and Daniel J. Sorin. Address translation aware memory
6397 consistency. *IEEE Micro*, 31(1):109–118, January 2011. doi:10.1109/MM.2010.99.
- 6398 [105] Naorin Hossain, Caroline Trippel, and Margaret Martonosi. Transform: Formally specifying
6399 transistency models and synthesizing enhanced litmus tests. In *47th ACM/IEEE Annual International*
6400 *Symposium on Computer Architecture, ISCA 2020, Valencia, Spain, May 30 - June 3, 2020*, pages
6401 874–887. IEEE, 2020. doi:10.1109/ISCA45697.2020.00076.
- 6402 [106] Runzhou Tao, Jianan Yao, Xupeng Li, Shih-Wei Li, Jason Nieh, and Ronghui Gu. Formal verification
6403 of a multiprocessor hypervisor on arm relaxed memory hardware. In *SOSP 2021: Proceedings of the*
6404 *28th ACM Symposium on Operating Systems Principles*, October 2021.
- 6405 [107] Jade Alglave, Richard Grisenthwaite, Artem Khyzha, Luc Maranget, and Nikos Nikoleris. Puss in
6406 Boots: Formalizing Arm’s Virtual Memory System Architecture. *IEEE Micro*, 44(6):83–91, 2024.
6407 doi:10.1109/MM.2024.3422668.
- 6408 [108] Jade Alglave, Luc Maranget, Paul E. McKenney, Andrea Parri, and Alan Stern. Frightening
6409 small children and disconcerting grown-ups: Concurrency in the linux kernel. In *Proceedings of*
6410 *the Twenty-Third International Conference on Architectural Support for Programming Languages*
6411 *and Operating Systems*, ASPLOS ’18, page 405418, New York, NY, USA, 2018. Association for
6412 Computing Machinery. doi:10.1145/3173162.3177156.
- 6413 [109] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*.
6414 Morgan Kaufmann, Amsterdam, 5 edition, 2012.
- 6415 [110] D. W. Anderson, F. J. Sparacio, and R. M. Tomasulo. The IBM System/360 Model 91: Machine
6416 philosophy and instruction-handling. *IBM Journal of Research and Development*, 11(1):8–24, 1967.
6417 doi:10.1147/rd.111.0008.
- 6418 [111] Alasdair Armstrong, Thomas Bauereiss, Brian Campbell, Shaked Flur Jon French Kathryn E. Gray,
6419 Gabriel Kerneis, Neel Krishnaswami, Prashanth Mundkur, Robert Norton-Wright, Christopher
6420 Pulte, Alastair Reid, Peter Sewell, Ian Stark, and Mark Wassell. Sail. <https://www.cl.cam.ac.uk/~pes20/sail/>, 2019.
- 6422 [112] Thomas Bauereiss, Brian Campbell, Alasdair Armstrong, Alastair Reid, Kathryn E. Gray, Anthony
6423 Fox, Peter Sewell, and Arm Limited. Sail Armv9.4-A instruction-set architecture (ISA) model, 2024.
6424 <https://github.com/rem-s-project/sail-arm>. Accessed 2024-05-11.
- 6425 [113] Allon Adir, Hagit Attiya, and Gil Shurek. Information-flow models for shared memory with an
6426 application to the PowerPC architecture. *IEEE Trans. Parallel Distrib. Syst.*, 14(5):502–515, 2003.
6427 doi:10.1109/TPDS.2003.1199067.
- 6428 [114] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John
6429 Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In
6430 *Proceedings of the 17th Annual International Symposium on Computer Architecture*, ISCA ’90, pages
6431 15–26, New York, NY, USA, 1990. ACM. doi:10.1145/325164.325102.

- 6432 [115] Kourosh Gharachorloo. *Memory Consistency Models for Shared-Memory Multiprocessors*. PhD
6433 thesis, Stanford University, 1995.
- 6434 [116] Pradeep S. Sindhu, Jean-Marc Frailong, and Michel Cekleov. *Formal Specification of Memory*
6435 *Models*, pages 25–41. Springer US, Boston, MA, 1992. doi:10.1007/978-1-4615-3604-8_2.
- 6436 [117] Intel. A formal specification of Intel Itanium processor family memory ordering, 2002. [http://
6437 //download.intel.com/design/Itanium/Downloads/25142901.pdf](http://download.intel.com/design/Itanium/Downloads/25142901.pdf).
- 6438 [118] Jade Alglave. *A Shared Memory Poetics*. PhD thesis, Université Paris 7 – Denis Diderot, November
6439 2010.
- 6440 [119] Luc Maranget. Personal communication, 2024.
- 6441 [120] Peter Sewell, Christopher Pulte, Shaked Flur, Mark Batty, Luc Maranget, and Alasdair Armstrong.
6442 Multicore semantics: Making sense of relaxed memory (MPhil slides), October 2022. URL: [https://
6443 //www.cl.cam.ac.uk/~pes20/slides-acs-2022.pdf](https://www.cl.cam.ac.uk/~pes20/slides-acs-2022.pdf).
- 6444 [121] Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, and Derek Dreyer. Repairing sequential
6445 consistency in C/C++11. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming*
6446 *Language Design and Implementation*, PLDI 2017, page 618632, New York, NY, USA, 2017.
6447 Association for Computing Machinery. doi:10.1145/3062341.3062352.
- 6448 [122] Mark Batty, Kayvan Memarian, Kyndylan Nienhuis, Jean Pichon-Pharabod, and Peter Sewell. The
6449 problem of programming language concurrency semantics. In *Programming Languages and Systems*
6450 *- 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint*
6451 *Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015.*
6452 *Proceedings*, pages 283–307, 2015. doi:10.1007/978-3-662-46669-8_12.
- 6453 [123] Michalis Kokologiannakis, Ori Lahav, Konstantinos Sagonas, and Viktor Vafeiadis. Effective stateless
6454 model checking for C/C++ concurrency. *Proc. ACM Program. Lang.*, 2(POPL), December 2017.
6455 doi:10.1145/3158105.
- 6456 [124] Michalis Kokologiannakis, Azalea Raad, and Viktor Vafeiadis. Model checking for weakly consistent
6457 libraries. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design*
6458 *and Implementation*, PLDI 2019, page 96110, New York, NY, USA, 2019. Association for Computing
6459 Machinery. doi:10.1145/3314221.3314609.
- 6460 [125] Michalis Kokologiannakis and Viktor Vafeiadis. GenMC: A model checker for weak memory models.
6461 In Alexandra Silva and K. Rustan M. Leino, editors, *Computer Aided Verification*, pages 427–440,
6462 Cham, 2021. Springer International Publishing. doi:10.1007/978-3-030-81685-8_20.
- 6463 [126] William W. Pugh. Fixing the Java memory model. In Geoffrey C. Fox, Klaus E. Schauser, and Marc
6464 Snir, editors, *Proceedings of the ACM 1999 Conference on Java Grande, JAVA '99, San Francisco,*
6465 *CA, USA, June 12-14, 1999*, pages 89–98. ACM, 1999. doi:10.1145/304065.304106.
- 6466 [127] Jean Pichon-Pharabod and Peter Sewell. A concurrency semantics for relaxed atomics that permits
6467 optimisation and avoids thin-air executions. In Rastislav Bodík and Rupak Majumdar, editors,
6468 *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming*
6469 *Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 622–633. ACM,
6470 2016. doi:10.1145/2837614.2837616.
- 6471 [128] Jeehoon Kang, Chung-Kil Hur, Ori Lahav, Viktor Vafeiadis, and Derek Dreyer. A promising
6472 semantics for relaxed-memory concurrency. In *Proceedings of the 44th ACM SIGPLAN Symposium*
6473 *on Principles of Programming Languages*, POPL '17, page 175189, New York, NY, USA, 2017.
6474 Association for Computing Machinery. doi:10.1145/3009837.3009850.
- 6475 [129] Radha Jagadeesan, Alan Jeffrey, and James Riely. Pomsets with preconditions: a simple model
6476 of relaxed memory. *Proc. ACM Program. Lang.*, 4(OOPSLA):194:1–194:30, 2020. doi:10.1145/
6477 3428262.
- 6478 [130] Soham Chakraborty. *Correct Compilation of Relaxed Memory Concurrency*. PhD thesis, Kaiser-
6479 slautern University of Technology, Germany, 2019. URL: [https://kluedo.ub.rptu.de/frontdoor/
6480 index/index/docId/5697](https://kluedo.ub.rptu.de/frontdoor/index/index/docId/5697).

6481 [131] Nathan Chong, Tyler Sorensen, and John Wickerson. The semantics of transactions and weak
6482 memory in x86, Power, ARM, and C++. In *Proceedings of the 39th ACM SIGPLAN Conference on*
6483 *Programming Language Design and Implementation*, PLDI 2018, page 211225, New York, NY, USA,
6484 2018. Association for Computing Machinery. doi:10.1145/3192366.3192373.

6485